Harry Stewart

INTER-OFFICE MEMO

TO: All Programmers, Marketing People, and others Interested in the DOS/FMS Reference Manual

FROM: Keith Brewster (108) DATE:

SUBJECT: PROPOSED OUTLINE & CONTENTS OF ATARI DOS/FMS REFERENCE MANUAL

1. The attached outline is based on my own meager knowledge of the DOS/FMS system, and reflects much personal opinion. Obviously, much material on the DOS (especially the BASIC commands/statements) have already been covered in our BASIC REFERENCE MANUAL, but for reader convenience, I plan to repeat most of this in the DOS book.

2. The concept of the order and format of this book is based on my knowledge and assumptions as a personal computer user: The new user will use the disk drive primarily as a speeded up method of saving and loading his programs, and not primarily as a method of storing data. Of course the more advanced owners will reverse this trend, and both usages must be thoroughly covered. This explains why I have chosen to "lead off" the body of the text with info on program saving and loading.

3. I am requesting ALL of you to review the outline very carefully and to send me any comments or ideas you may have as soon as humanly possible so that I can get work on the book underway. Without YOUR input, I am put in the position of having to attempt to read minds, and I have proved to my own satisfaction (and yours, I suspect), that this is beyond my somewhat limited talents. Since time is now a very real factor in the preparation of this book, if I do not receive your comments by Monday afternoon (Sep. 24, 1979), I cannot be responsible for later revision of the book to include subjects that I have ommitted in your opinion!

4. Technical writers are neither engineers nor programmers, although we sometimes try to be both...We are "word merchants"...translators who must write in English what folks like you tell us to. We are utterly at the mercy of those of you who know what you are talking about, for we seldom do! We MUST and DO depend on the information and ideas you provide us. But if you wait until the book is already written, it may be too late!

HELP! HELP! HELP!

And Thanks!

TENTATIVE OUTLINE OF DOS/FMS OWNERS' REFERENCE MANUAL

PREFACE

CHAPTER 1. Introduction to disk drives...How to use this Manual (with flowchart)...Definitions of common terminology...Format and conventions used in descriptions of Disk BASIC statements...A few things you can do with your ATARI disk system.

CHAPTER 2. Commonly used Disk Procedures: Step by step instructions on:
- A. Duplicating your master diskette.
- B. Saving programs on diskettes.
- C. "Downloading" (into RAM) programs stored on diskettes.

CHAPTER 3. What can your disk drive do for you? More specific explanations and information.
- A. Saving and loading programs.
- B. Storing and retrieving data.

CHAPTER 4. Back to BASICs: BASIC statements used in Disk Operations.
- A. NOTE and POINT
- B. PUT and GET
- C. OPEN and CLOSE
- D. INPUT and PRINT
- E. SAVE and LOAD
- F. ENTER
- G. RUN and LIST
- H. XIO
- I.
- J.
- K.
- L.

(PLEASE FILL IN ANY BLANKS)

CHAPTER 5. Data Files on Disk - The FMS
- A. Introduction to Data Files
- B. Sequential Access Files
- C. Random Access Files.
- D. End of File Markers
- E. "Filespecs": How to name your Files.

Note: DO WE HAVE THESE? HOW ARE THEY USED?

CHAPTER 6. The Disk Operating System and Its Commands
- A. The Disk Directory
- B. The Run Cartridge command
- C. The Copy File command
- D. The Delete File command
- E. The Rename File command
- F. The Lock File command
- G. The Unlock File command
- H. The Write DOS File command
- I. The Format Disk command
- J. The Duplicate Disk command
- K. The Binary Save command
- L. The Binary Load command
- M. The Run Address command.
- N. The Define Device command

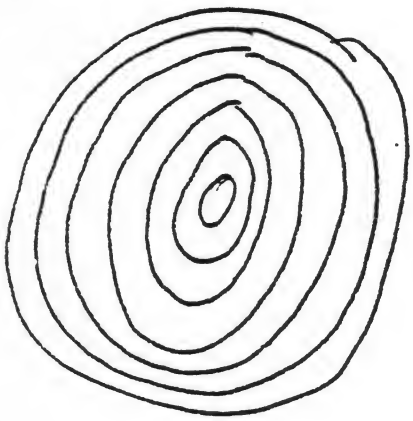
APPENDICES: These will cover diskette specifications, DOS/FMS error messages, BASIC disk statements and reserved words, the DOS commands, and the Glossary/Index.

ATARI

800/810

DOS MANUAL

SOFTWARE CONTROL FOR
YOUR ATARI 810 DISK DRIVE...



– Keith Brewster – (Output)
– Ian Sheppard – (Input)

PREFACE ~~A Personal Note from the Author~~

Welcome to the expanded computing world made possible by your purchase of an ATARI Diskette system! You will soon learn to appreciate the rapid-access storage facility you now own. It will give you many kilobytes of data or program storage that is yours to use at the touch of a few keys. But which keys, and in what order? That, dear customer, is the subject of this book!

Like the BASIC Reference Manual, the style is light and somewhat humorous. If I can entertain you enough to keep you reading, then at least there is a chance that you will learn. Boring, dry books eliminate that chance!

We at ATARI know that there are three large classes of new disk drive owners: First, there are those of you who have purchased your drive along with the rest of your system, and may have no knowledge of computers at all. If you fall into that category, please take our advice and read the BASIC Reference Manual first. Once you have become the master of your system without the disk drive plugged in, then come back amd use the drive and this book.

The second large category of disk-users consists of those who have had some experience with the ATARI computer systems and who are now "upgrading" them to include disk capabilities. These folks are already familiar with the BASIC Reference Manual, and should now proceed to Chapter 1 of this book. Of course a short review of the BRM Chapter 5 can't hurt a thing....

In the third category are those of you who have significant computer experience but who don't yet know the ATARI departures from "standard" BASIC. The Disk Operating System (DOS) and its File Management Software (FMS) must now be learned. The best advice we can offer you folks is to plug the disk-drive into your system and learn its use as you work with the rest of the system.

Chapter 2 will introduce you to the simple, very popular disk operations (duplicating the DOS software on new diskettes, SAVEing files on diskette, and LOADing these back into RAM.)

You may be interested in knowing that with the exception of the cassette Program Recorder, all ATARI peripherals are "smart"...that is they have microcomputer chips built right into them. Of course this is true of your new 810 Disk Drive.

We're sure, too, that you are curious about the capabilities of the disk system. At this introductory level, let's just say that you can think of a diskette as auxiliary RAM memory, capable of storing data and programs (both of which are referred to as "files" in computerese.) Like RAM, the disk system allows virtually instantaneous and ~~nearly~~ "random" access to the data. This is briefly explained in the 810 Disk Drive Owner Handbook, which will also tell you how to plug in and load your drive(s).

Capabilites of any disk drive system are dependent on a series of compromises! The more varying capabilities they have, the more complex they are to use. WE like to think that ATARI has optimized these two conflicting requirements for the home and small business user.

Whatever your disk requirements and your type and level of experience, we hope you will enjoy this book! We've tried our best to make it entertaining and fun to read as well as technically accurate and complete.

Welcome to ATARI's Disk Operation System...

- Keith Brewster -

- Ian Sheppard -

DOS MANUAL

CHAPTER 1 - INTRODUCTION TO DISK OPERATIONS

1.1 Introductory Information

This Chapter will help you find your way around both your new disk system and this Manual. It will not tell you how to plug it in or fire it up. That's what the Owner's Manual is for. What it will tell you is how to control it by typing messages to the computer, for that is the function of software, and this is a book about pure software. In case you've forgotten, or if you are a new computer user, software is computerese for control of a computer or accessories that is not a physical mechanism or electronic circuit, but which consists entirely of electronic "states" or signals as controlled by the human operator or programmer. Any program is software. Your disk operating system (DOS) is a collection of closely related programs that allow you to control your disk drive.

If you just can't wait (you may have a program in your computer that you entered after you connected your new drive system), there are a few very simple operations that you can conduct easily without further study. These include SAVEing and LOADing programs on diskettes, and preparing a new (totally blank) diskette for your use in the ATARI system. If you're in a hurry, simply "GOTO CHAPTER 2". But do come back later!

While we are introducing new concepts and finding our way around, let's take a quick time out for a brief discussion of diskettes, what they are really like, and how to care for and store them properly. It is very difficult to convey to you just how important proper care of your diskettes is...let's just say that nothing is really much more important. If you get a program all snarled up, you can always try again. If you ruin a disk, you may lose 30 programs or a whole bunch of irreplaceable data.... So take the time right now to read the inside front cover of this Manual for some tips on proper care of your valuable diskettes!

1.2 Some Simple Diskette and DOS Terminology

Appendix of this Manual contains an extensive Glossary/Index of the many terms and "computerese" words that are often used when describing disk-oriented hardware and software. At this point, let's just look at a few of the most common ones:

Diskette: A diskette is simply a small floppy disk! The word "floppy" has become popular in the computer industry to distinguish between flexible and the old-style rigid disks. Large disks are often 8 inches or even larger in diameter. Diskettes are only 5 inches across. This allows the size of the disk drive to be reduced, and both convenience and economy are improved. Diskettes are made of plastic material coated with a special magnetic material very similar to that used on magnetic recording tape.

DOS: Pronounced "doss", this is an acronymn for disk operating system. DOS is a collection of programs (software, right?) used by the computer and the drive (which is "intelligent"...that is, it has a microcomputer chip built into its internal circuitry) to allow the total system to do what we tell it to do (and just as important, not to do anything else!) DOS is directly related to OS, the operating system "burned" into ROM within the computer itself. In fact, it is really an extension of the OS. DOS includes the file management system (called FMS, naturally!) which is the portion of the software that helps locate and "manage" the data we store on diskette.

Menu: With a disk installed (that has the proper DOS on it), when the system is powered up by switching on the computer with the disk drive already on, you will hear a series of clicks and beeps and other interesting noises that tell you that the disk drive is doing something. About 5 seconds later, the prompt READY will appear on the screen. If you then type DOS (followed by RETURN as usual), you will see what computer folks call the MENU. Just as a restaurant menu does, the computer menu displays a series

of selections you can make. Unfortunately, they aren't very tasty, but they certainly are useful! Any time you want to see the menu, just type DOS followed by RETURN.

Directory: Similar to the menu, the directory is really somewhat more like the table of contents in a book. It displays a list of the files on the particular diskette you had loaded when you turned on the computer. To see the directory, get the menu by typing DOS, then type A (the letter next to the word Directory on the menu) and RETURN. The computer will then ask you whether you want it to scan all entries in the directory or whether you'd rather enter a particular file specification. Since you want the whole thing, just press RETURN again. More on this later! Much more!

SAVE: A BASIC command that lets you output the contents of RAM to the diskette.

LOAD: A BASIC command that inputs the data on the disk into the computer's RAM.

Boot: Remember we said that DOS is really an extention of the ROM contained computer operating system (OS)? The term "boot" is a reference to the old expression "by its own bootstraps" and means that the diskette contains a program that allows the disk system to run programs, but it needs a program to do that so the diskette has a program... When we power up a computer with a disk system, the disk is said to "boot" or "boot up". If something goes wrong during this process, the words "BOOT ERROR" are displayed on the screen. In this case, power down just the computer, wait about 5 seconds, then power up again.

Format: We'll give details later, but for now, think of a diskette as a flat piece of plactic that will become a contentional phonograph record. It has no music on it, and no "grooves". We can think of the process of formatting as cutting some magnetic "grooves" that will help the computer/drive find its way

around on the diskette later. The process of formatting is done at the same time as a new diskette is having the DOS magnetically recorded on it. Remember that before you can use a new blank diskette, you must first format and record the DOS on it. Without DOS you don't have a disk drive!
See Chapter 2 for a description of the formatting/DOS recording procedure. It's really very easy, for your computer does all the work...

File: A file is just a collection of related data (data is computerese for information). A file may be a list of phone numbers, checks, addresses, or a complete BASIC program. The word file is a very general, loosely used one. In this book, we will restrict its use to mean a collection of data or a program that has been or will be stored on a diskette.

1.3 Conventions Used in this Manual

All of the terms, usages, and conventions incorporated in the BASIC Reference Manual will be used in exactly the same way in this book. To save the reader the cross-referencing efforts, the section from the BASIC Reference Manual on terms and conventions is reproduced on the next few pages.

The following conventions are applied throughout this Manual. If the reader will take a little time before proceeding to LEARN THE CONVENTIONS, he will have MUCH less trouble understanding some of the format explanations used throughout the rest of the text.

CAPITAL LETTERS: In this book, CAPITAL LETTERS denote keywords which must be typed by the user in upper case form exactly as they are printed in this text. Reverse-video characters will not work. Here are a few examples:

PRINT RUN LIST END GOTO GOSUB FOR NEXT IF

Lower Case Letters: In this manual, lower case letters are used to denote the various classes of items which may be used in a program, such as variables (var), expressions (exp), and the like. The abbreviations used for these classes of items are shown in Table 1.1 below.

Items in Brackets: Square brackets like these [ ] are optional items which may be used but are not required. These brackets must not be confused with parentheses like these ( ) which are entirely different!

Items in Brackets followed by Three Dots [exp...]: These items are optional, but may be used in any number desired. Thus [, exp...] is just the same as [, exp, exp, exp] and so on. The dots just mean (in this case) that any number of expressions may be "tacked on" as needed, but none are required.

Items stacked vertically in braces: Items stacked vertically in braces like these { item / item } mean that any one of the stacked items may be used, but that only one at a time is permissible. Here's a good example of this:

100 { GOTO / GOSUB } 2000

(Here either GOTO or GOSUB may be used, but not both!)

TABLE 1.1

ABBREVIATIONS USED IN THIS MANUAL

ABBREVIATIONS	EXPLANATION AND DISCUSSION OF ITEMS
avar	Arithmetic Variable: A location where a numeric value is stored. Variable names may be from 1 to 120 alphanumeric characters, but must start with an alphabetic character, and all alpha characters must be unreversed, upper case.
svar	String Variable: A location where a string of characters (bytes) may be stored. Same name rules as avar except that the last character in the variable name must be a $.
mvar	Matrix Variable: (Also called a subscripted variable). An element of an array or matrix. The variable name for the array or matrix as a whole may be any legal variable name such as A, X, Y, ZIP, K, or whatever. The subscripted variable (name for the particular element) starts with the matrix variable, and then uses a number, variable, or expression in parentheses immediately following (without a space!) the array or matrix variable. For example, A(ROW) A(1), A(X+1).
var	Variable: Any variable. May be mvar, avar, or svar.
aexp	Arithmetic Expression: Generally composed of "aexp aop aexp" where each aexp may be an lexp, avar, mvar, constant, or arithmetic function. Any arithmetic expression...
lexp	Logical Expression: Generally composed of aexp lop aexp or sexp lop sexp. Such an expression evaluates to either a 1 (logical true) or a 0 (logical false).

Table 1.1 (Continued)

ABBREVIATIONS	EXPLANATION AND DISCUSSION OF ITEMS
lexp	For example, the expression 1<2 evaluates to the value "1" (true) while the expression "LEMON" = "ORANGE" evaluates to a ~~zero~~ "0" (false) as the two strings are not equal.
sexp	String expression: Can consist of a string variable, string literal (constant), or a function that returns a string value. No operators are allowed in sexp.
aop	The arithmetic operator signs: +, -, *, /, <, <=, =, >=, <>, ^
lop	The logical operators: AND, OR, NOT (the latter is a special case: NOT is a unary operator)
lineno	A constant that identifies a particular program line in a deferred mode BASIC program.
exp	Any expression, whether sexp or aexp.
adata	Data expressed in ATASCII code. Quotation marks are "thrown out" by the computer, as are leading blanks. (see Chapter 5 and Appendix E)
"filespec"	Reference to a particular file, usually on disk. "file spec" always appears in quotes, and contains information on the type of I/O device, its number, a colon, an optional file name, and an optional filename extender. Example filespec: "D1:NATALIE.ED"

1.4 Examples of Disk Capabilities:

A peek into Appendix will reveal some of the specifications of your disk drive and diskettes. As you can see, each diskette can store an incredible volume of data, and this gigantic collection of 1's and 0's is kept "straight" by the FMS program within DOS. You can SAVE programs on diskettes and LOAD them back into RAM for execution in a very few seconds. Here are a few typical examples of what you can do with your disk system. As seems to be typical of most computer operations, the real limitations are within the user's own mind. Remember that mathematics can represent anything in the real world, that binary operations can do any mathematical operations, and that electronic circuits can do any binary mathematics! It should be obvious from these three truths that our electronic computers can do (or at least represent) anything!

How about storing phone numbers of all your friends and relatives. Using simple computerized sorting routines, these can be placed in alphabetical order, sorted by area code,..whatever your mind can conceive! How 'bout something similar for addresses, sorted by zip code for mass mailing? Want to keep track of every check you write this year for income tax purposes? Then when the IRS asks questions, you can have their computer speak to yours! (Just a joke, today, folks...but it's coming!) How about a menu planner based on the favorite items of your guests? A household budget can be a practical program...if you have disk data capabilities! And what about your Christmas card list? Use your imagination. Come up with your own ideas. Then use your BASIC Reference Manual and this DOS Manual to make your "pipedreams" come true! We're always here to help.

CHAPTER 2: FORMATTING, SAVE, AND LOAD USING DISKETTES

FIGURE 2.1 - THE MENU (TYPE DOS TO SEE THIS)

```
DISK OPERATING SYSTEM                          9/24/79
COPYRIGHT 1979 ATARI

A. DISK DIRECTORY                  I. FORMAT DISK
B. RUN CARTRIDGE                   J. DUPLICATE DISK
C. COPY FILE                       K. BINARY SAVE
D. DELETE FILE(S)                  L. BINARY LOAD
E. RENAME FILE                     M. RUN AT ADDRESS
F. LOCK FILE                       N. DEFINE DEVICE
G. UNLOCK FILE                     O. DUPLICATE FILE
H. WRITE DOS FILE

SELECT ITEM
```

Note: Command A (DISK DIRECTORY) is very helpful if you forget a file name. Just type A [RETURN]. The computer will ask you for a filespec. Don't type anything, but just hit [RETURN] again. The directory (like a table of contents) will be displayed and you can then find your filename. ~~Press~~ ~~Type~~ DOS [RETURN] to get back to Menu.

2.1 Introduction to SAVEing and LOADing Programs and Formatting New Diskettes:

This Chapter will help you more experienced readers (and those of you who are in an insurmountable hurry to try out your new equipment) conduct the most basic operations possible with the ATARI disk system. It will first tell you how to make copies of your Master Diskette (the one supplied that has DOS on it), formatting a new diskette in the process, how to SAVE a program on diskette, and how to LOAD the SAVEd program back into your computer.

It is very important that you do not use your Master DOS Diskette for any purpose other than to format new diskettes (including storing the DOS software on them). As we've said before, but won't again after this one last attempt to drive this point home: Without DOS, you don't have a disk system!

2.2 Formatting a New Disk:

Just a few steps will accomplish this. Note that the same steps may be used if you want to "recycle" an old disk, erasing what is on it and reformatting for further use. Here's the procedure, step by step. There is also a "flowchart" version of these (same) steps in Figure 2.2, so use what works best for you!

Step 1: Load your Master Diskette into Drive 1 with drive power on, and ~~all power off. Now connect power to the drive unit only.~~ console off.

Step 2: Now power up the computer console. Within about 10 seconds, it will display the READY prompt which means that DOS has been properly "booted up".

Step 3: Remove your Master Diskette and replace it in its envelope. Load a blank diskette into the drive carefully.

NOTE: Diskettes should never be inserted or removed from drive with drive latch lowered down.

Step 4: Type the word DOS and hit RETURN. You will see the menu displayed on the screen. This should look like Figure 1.1. The words SELECT ITEM will be displayed below the menu as shown in the Figure.

Step 5: Note that the letter I is next to the menu item "FORMAT DISK". Select I by simply typing that letter followed by RETURN. The computer will reply with "WHICH DRIVE TO FORMAT?"

Step 6: Since we are using drive #1, type the number 1 (and hit RETURN). The computer will come right back with TYPE "Y" TO FORMAT DISK 1.

Step 7: Hey! Who's the boss here? We're supposed to tell it whatto do next, right? Oh well, in this case it knows what its doing and we don't, so let's just humor it a while...Type the letter Y and hit RETURN. It will honk, beep, and click for about 30 seconds, and then it will come up with SELECT ITEM again.

Step 8: Look at the menu, and note that option H is next to the item called "WRITE DOS FILE", which is what we want to do. Type the letter H and hit RETURN again. It will come up with TYPE "Y" TO WRITE NEW DOS FILE.

Step 9: Again, let's humor it! Type the letter Y and RETURN. It will answer us by saying WRITING NEW DOS.SYS FILE. Again it will honk and click and bump and grind for about 30 seconds. Finally, it will tell us to SELECT ITEM once again. The job is done. Our new disk is now formatted, and has a copy of the DOS file on it.

Step 10: Just to make sure, power the computer down, (erasing the DOS file in RAM), and then power up and let the system "boot" under the control of the new diskette. If it boots ok, it will soon come up with READY which tells us our format/copy job has "taken" perfectly.

Figure 2.2

PROCEDURE FOR FORMATTING AND COPYING DOS

Each time a new (blank) diskette is to be used, it must be formatted and the DOS software copied onto it. To do this, proceed as shown in this flowchart. Use Drive #1 for all formatting and DOS copying.

WHEN THE COMPUTER SAYS THIS	YOU DO THIS		IT WILL DISPLAY THIS
	Load MASTER DISKETTE into Drive 1		
	Power console up: DOS will "Boot up"		
	Remove MASTER, load BLANK diskette		
	Type "DOS"	RETURN	MENU of operator choices. SELECT ITEM
MENU of operator choices. SELECT ITEM	Type "I"	RETURN	WHICH DRIVE TO FORMAT?
WHICH DRIVE TO FORMAT?	Type "1"	RETURN	TYPE "Y" TO FORMAT DISK 1
TYPE "Y" TO FORMAT DISK 1	TYPE "Y"	RETURN	After about 30 seconds: SELECT ITEM
SELECT ITEM	TYPE "H"	RETURN	TYPE "Y" TO WRITE NEW DOS FILE
TYPE "Y" TO WRITE NEW DOS FILE	TYPE "Y"	RETURN	WRITING NEW DOS. SYS FILE
	After about 30 seconds, computer will display "SELECT ITEM"		

NOTES:

1. In this procedure, it is never necessary to key quotation marks into the system.
2. Never load or unload a diskette into the drive with the drive busy light on or with power to the drive shut off. The computer console may be powered down, however.
3. It is good practice to keep a spare copy of the DOS on a diskette just in case something "happens" to your original MASTER diskette. Without DOS you haven't got a disk-drive system!

2.3 How To SAVE a Program on Diskette:

Before we give you the simple instructions covering the SAVEing of your programs on diskettes, there are a few things that you must understand. First of all, you must "boot" a disk when you power up. This is because once a program is in RAM, it will be lost any time power is shot off. As we've already seen, we must power down the computer and then power it up with the disk with DOS on it in order to make use of our disk system! It is good ~~practice~~ policy to always boot up whenever you use your computer system, making sure that the diskette in the drive has sufficient capacity to store as much data as your system RAM can. How to do this ?

Secondly, it is not necessary that data to be stored on diskette come from the keyboard! Any program in RAM can be SAVEd. This means you can now transfer your cassette programs to diskette for increased LOADing speed. SimplyCLOAD from cassette in the normal way (see the BASIC REference Manual), then SAVE onto disk as described below.

Thirdly, you must have a grounding in how file specs and file names work. To create a legal filename, you may use any eight characters, provided that

1. All eight are either capital letters or numerical digits.
2. The first character is a letter.
3. No reverse video characters, special characters (graphics, %, etc) or spaces are used.

Thus COLLEEN is a legal file name (even though there are only 7 letters.... eight is a maximum, not a required number). The eight (maximum) characters may be optionally followed by a period and up to three more characters. This is called the "extender". The three characters in the extender may be either capital letters or numerical digits in any combination. Here are a few more examples of legal and illegal file names:

ATARI.BAS (Perfectly legal)

1ATARI.111 (Illegal: First character must be a letter)

ATARI22.XYZ (Legal name)

A1234567,889 (Legal name)

A ATARI.BAS (Nope! No spaces allowed. Capital letters and numbers only!)

Got the idea? Good! Note the filename for the ATARI DOS is DOS.SYS, which of course is legal. So much for file names. Now for what we call file specifications or "filespecs".

A filespec as used with the SAVE command is of a particular short format. ~~Those of you who have carefully read Chapter 5 of the BASIC Reference Manual will understand when we say that the OPEN statement is included in the SAVE statement.~~ SAVE is a BASIC keyword that is kind of a shortcut to disk program storage. But back to filespecs! The filespec (as used as a BASIC statement component) is always in quotation marks, and starts with the letter D followed immediately by a colon and then the file name. The format of a typical command to SAVE a program looks like this:

SAVE"D:filename"

Let's say, for example, we have written a program to play poker. We want to SAVE our program on disk. First we must pick a legal filename, and a good choice might be POKER.BAS (the BAS indicates that the program is in BASIC). Here is what we would tell the computer:

SAVE"D:POKER.BAS" RETURN

The disk drive will honk and beep and click for a few seconds (very few) and the word "READY" will appear in the normal manner. Beats "cuing up" a cassette, huh? Quick, too!

2.4 LOADing a Diskette Program Back Into RAM:

Riddle: What's even easier than SAVEing a program on diskette? Answer: LOADing it back into RAM. Simply tell it to LOAD"D:filename". In our poker game example, it would look like this:

LOAD"D:POKER.BAS" RETURN

In a second or three your program will be loaded into RAM as indicated by the READY prompt. Now you simply RUN it in the normal way...

This concludes our chapter on "quick and dirty" disk operations! Just to whet your appetite in hopes that you will (someday) read the rest of this book, try this command in Direct Mode: (Have your POKER or whatever disk in place, the drive on, and then "boot" the disk by turning on the console. Now type this:

RUN"D:POKER.BAS" RETURN

Or you can:

LIST"D:POKER.BAS" RETURN

There's more, much more, to disk operation than we've showed you so far! Things get better and better. In fact, a good disk system in the hands of an expert (which you will soon be...) is worth many many times as much as one without the disk equipment. You'll see...

CHAPTER 3: DISK OPERATIONS USING ATARI BASIC

As you may remember from your studies of the BASIC Reference Manual, disk operations are a type of input/output (I/O) technique, just as are keyboard, Program Recorder, TV Screen, Screen Editor, Printer, ~~and other operations involving parts of the total system other than the CPU.~~ All I/O operations are performed using the same simple technique: First a file is opened using a device code, a reference number, and other file specifications and names. Second, the information is copied ~~into or out of~~ the computer console (CPU). Finally, the file is closed (using the reference number established when it was opened). All of these steps can be accomplished using BASIC statements, and no esotetic or complex machine-level codes or POKEs need be used.

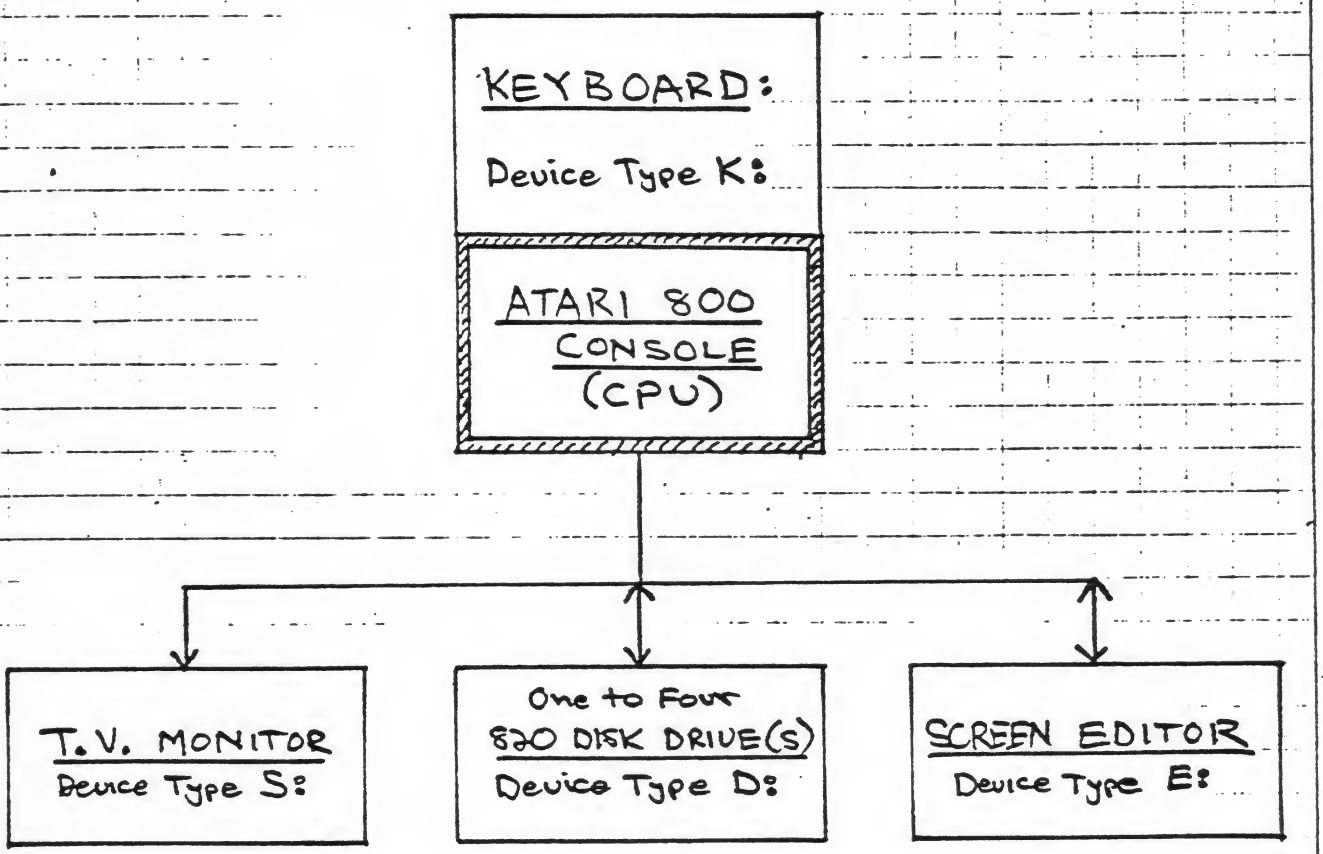
Some of the most used disk techniques have been made even simpler by the provision of "shortcut" statements (like SAVE and LOAD) which do not require separate opening and closing of files. Others have been incorporated into the DOS itself so that they can be accomplished by selecting items from the menu displayed each time "DOS" is typed followed by RETURN. Some of these are DELETE FILE, DUPLICATE DISK, COPY FILE, FORMAT DISK, and RENAME FILE. We'll discuss each of these in detail in Chapter 5. We've already seen SAVE and LOAD at work in Chapter 2. This Chapter will cover all of the BASIC keywords used to control disk drive procedures. Any time you're ready, let's press on! (That's the switch on the side of your computer!)

3.1 Steps in Preparation for Disk Equipped Computer Work:

Anytime you are preparing to write a program or enter one into your computer from the keyboard or the Program Recorder, connect your disk drive, turn it on, and load a diskette with plenty of free space to record your program and data files (if used). After this is done, "boot" the system by switching on the computer. When the clicking and whirring has stopped, and the word "READY"

Figure 3.1

DISK ("D:) I/O OPERATIONS – BLOCK DIAGRAM



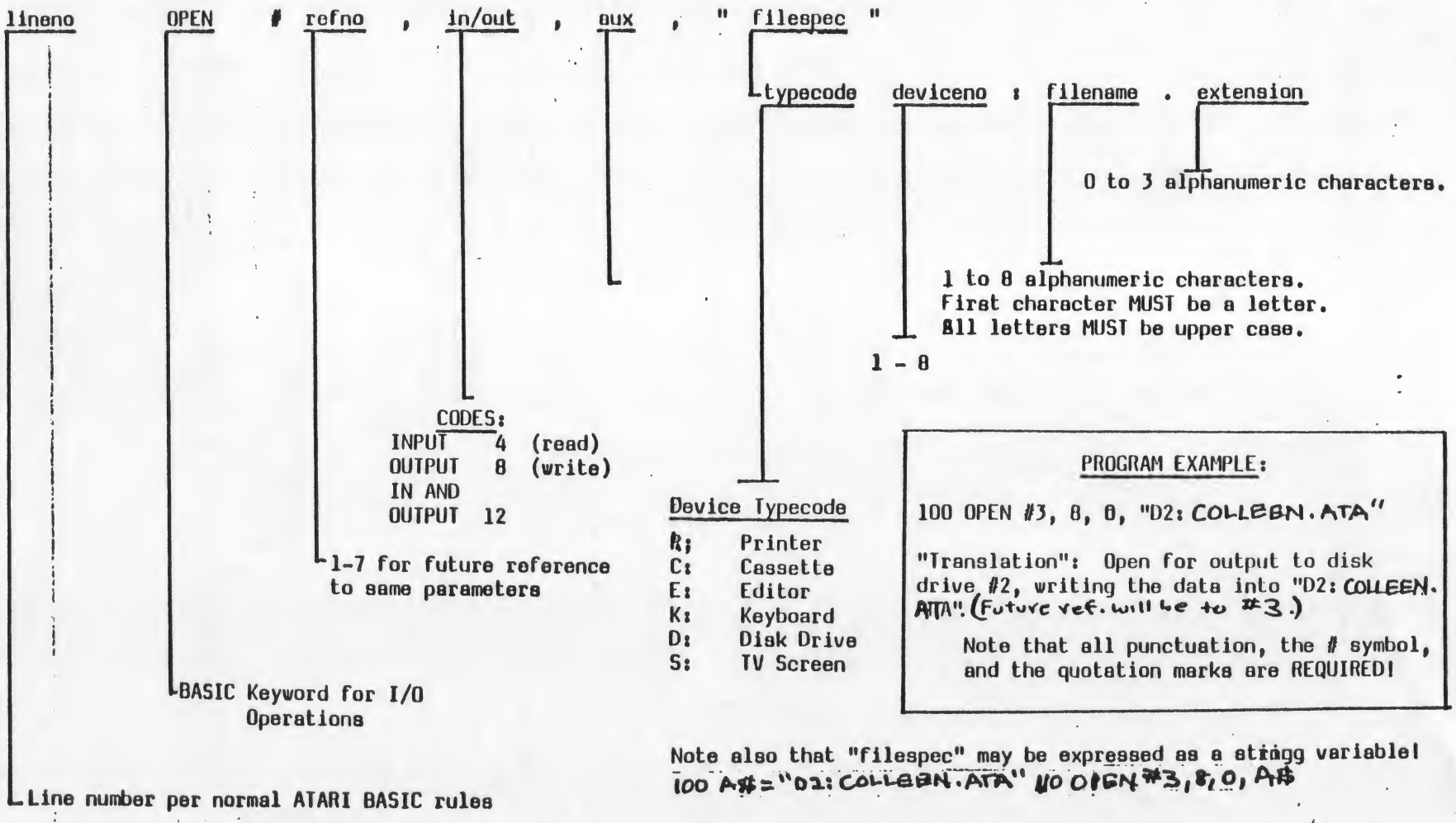
DISK I/O KEYWORDS

OPEN, CLOSE, LIST, ENTER, SAVE, LOAD, NOTE, POINT, PUT, GET, XIO

Figure 3.2
(Revise to be same as BRM final version)

NOTE: For purposes of clarity, we have temporarily abandoned the conventions described in Chapter 1. The reasons for this will become obvious, for the OPEN statement is a very general Input/Output statement and has many options and may take many apparently different forms.

FIGURE 5.2 - THE OPEN STATEMENT - GENERAL FORMAT



PROGRAM EXAMPLE:

100 OPEN #3, 8, 0, "D2:COLLEEN.ATA"

"Translation": Open for output to disk drive #2, writing the data into "D2:COLLEEN.ATA". (Future ref. will be to #3.)

Note that all punctuation, the # symbol, and the quotation marks are REQUIRED!

Note also that "filespec" may be expressed as a stringg variable!
100 A$="D2:COLLEEN.ATA" :OPEN #3,8,0,A$

appears in the upper left hand corner of your screen, programming or entering of programs may begin. (But first, make sure that there is nothing in memory. Type NEW followed by RETURN.)

Now we're ready to look at the BASIC keywords used with disk operations, along with some examples of how they are used. Here goes:

OPEN:

Format: lineno OPEN # aexp , aexp1 , aexp2 , "filespec"
(note that "filespec" may be a string variable!)

Program Example: 100 OPEN #1, 4, 0, "D1:SOFTWARE.BAS"

Notes:

1. lineno is the normal ATARI BASIC line number.
2. OPEN is the BASIC keyword.
3. # is a mandatory character
4. aexp is an expression that evaluates to the number that will be used for future reference and indicates the same set of parameters. It will be used later with CLOSE.
5. aexp1 may evaluate to 4, 8, or 12. 4 means an input (toward the computer) transfer of data will take place, 8 means the transfer will be an output to the disk drive from the computer, and 12 means that either input or output (or both) may be in the offing. (The opposite of onning?)
6. aexp2 is an auxiliary printer control code and is not used with disks. A 0 must, however, be used (like a dummy variable.)
7. filespec is a specific file designation. The quotation marks around it are required unless a string variable is used. (See example below).
8. The commas (see format and program example above) must also be typed into the computer.

9. filespec consists of the device code (D: for disk) along with a number that indicates which of the particular devices of this code will be used, followed by a required colon (:) followed by the filename and extender. For example:

```
"D3:ATARI.800"          (a legal filespec)
        or
100 A$="D3:ATARI.800"   (store filespec  in variable A$)
110 OPEN #1, 8, 0, A$   (refer to filespec as variable A$)
```

Here are two more program examples with "translations":

100 OPEN #3,4,0,"D1:BUDGET.BAS" Open for input to the computer, the file designated "D1:BUDGET.BAS", and assign a future reference number of 3. Now the number #3 can be used to refer to this particular OPEN statement when doing GETS, INPUTS, etc. Disk drive #1 is specified in the filespec.

100 OPEN #1,8,0,"D3:CHECKBKK.BAS" Open for output (the number 8) to the disk file called "D3:CHECKBØK.BAS" (disk number 3) and assign the future reference number #1.

In summary, we can say that the OPEN statement is used to first establish a reference number so we won't have to go through the whole I/O routine again each time we conduct an operation, second to tell the computer whether input or output (or both) operations may be expected, establishes the type of device (disk, in this case) is to be used, and which one if more than one disk drive is connected, and finally tells the computer what the file name is.

Important Note: Several I/O operations may be OPEN at the same time!

CLOSE

Format: lineno CLOSE # aexp

Program Example: 100 CLOSE #2

CLOSE simply CLOSEs files that have previously been OPENed. aexp may be a number, a variable, or an arithmetic expression, but must evaluate to a number corresponding to an OPEN command used earlier in the program. For example, if we have previously done this:

```
100 OPEN #3, 4, 0, "DISKSOFT.BAS"
```

We can now continue with the input operation that has been OPENed. When the last operation has been completed, we must CLOSE the I/O file with the command :

```
200 CLOSE #3
```

Now we can readily see what that first aexp (the one following the #) in the OPEN statement was really good for! It saves us having to go through the whole routine for CLOSE that we did for OPEN.

NEW, END, and RUN perform the CLOSE operation automatically, and once a file has been CLOSEd, the OPEN statement is completely "forgotten" by the computer. If the same I/O parameters are to be used again, the file must be re-OPENed using the identical procedure that was used the first time it was OPENed. Of course, a different reference number may be used "the second time around" if desired. This is not necessary, though.

Once we have an I/O "file" OPENed, we can make use of many of the other disk drive statements available to us in ATARI basic. The following section of this Manual disausses these:

NOTE and POINT

Format: lineno NOTE # aexp, avar, avar

what are the range of value

lineno POINT # aexp, avar, avar

Program Examples: 100 NOTE #1, X, Y

200 POINT #4, A, B

NOTE is used to store the current disk sector number and the current byte number. aexp evaluates to the I/O reference number, the first avar is used to store the current sector number, while the second avar holds the current byte number. The result is the current read or write position in the specified file (where the next byte is located that will be read or written to.)

NOTE is usually used when writing data to a file (output from the computer). The NOTE information is written into a second I/O file, and is then used as an index to the first file, (one method of doing random access disk read/writes).

POINT is used when reading a file (input to the computer). It specifies to the computer the sector number (first avar) and byte (second avar) within the sector.which will next be read. That is, it moves a software controlled pointer to the specified location in the file. This is used for random access to the data stored in the disk file.

We'll see much more on Random Access files later in this Manual. But one method of handling these is to use NOTE and POINT. If we first OPEN our I/O operation, we can then NOTE the location where the next piece of data will be stored. By doing a PRINT # , we can cause our next data to be written to disk in the NOTEd location. To read the file, all we have to do is POINT, giving the location of the specific data we want to retrieve, and then INPUT # will bring it back into the computer. Details in Chapter _. The big thing to get in our minds is that NOTE and POINT both locate the sector and byte that will be accessed next (and not the one that has just been accessed!)

PUT and GET

Format: lineno PUT # aexp, aexp

lineno GET # aexp, avar

PUT is an output statement, while GET is for input of data from diskette to the computer. Both statements may be used to input/output a single byte of data at a time. aexp is the I/O reference number as usual. For PUT, the aexp following the reference number is the byte (or evaluates to the byte) being output. GET is exactly the opposite of PUT. It reads one byte (using the #aexp to designate the OPENed file), and then stores the byte read in the variable avar. PUT and GET are two of the most commonly used disk statements. Note that by using the ATASCII codes (see the Basic Reference Manual), characters may be input or output using PUT and GET. Since the one-byte capability of the statements PUT and GET can handle up to decimal 256, all ATASCII characters can be handled with these two statements!

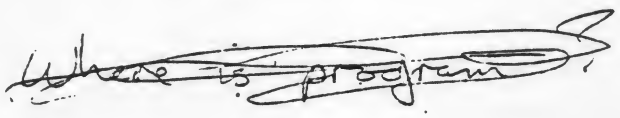
INPUT and PRINT

Format: lineno INPUT # aexp , var (may be svar or avar)

lineno PRINT # aexp , exp

Program Examples:

```
10 DIM A$(10), B$(30)
20 A$ = "disk input"              (note lower case ok in string)
30 OPEN #1,8,0,"D1:EXAMPLE$.BAS"
40 PRINT #1; A$
50 CLOSE #1
60 OPEN #3,4,0,"D1:EXAMPLE$.BAS"
70 INPUT #3;B$
80 CLOSE #3
90 PRINT "A$ IS ";A$
100 PRINT "B$ IS ";B$
```



As the above program example (try RUNning it!) shows, INPUT and PRINT are very much like PUT and GET except that they are not limited to single byte operations, but can handle virtually any I/O operation! There's only one tricky thing about PRINT: Use a semicolon (;) after PRINT #1; not a comma! If you use a comma, the I/O system will treat it just exactly as it would if you were PRINTing to the monitor screen! It will insert 10 blank spaces. "So what?" I hear you cry. "Who cares?" Well, let's look at it this way: Say you've sent out a ten character string to disk with PRINT, and now you go to get it back into RAM with INPUT. Since we've dimensioned the string to hold the 10 characters, ten characters will be INPUT. But all ten of 'em are spaces! Now who cares? So let's just remember to use a semicolon instead of a comma with PRINT #. Unless, of course, we want to put a string of blanks on the screen in front of our character string...in that case, we must be careful to DIMension our string to hold both the characters and the ten zeros added by the comma!

If you want to see all this in action, go back to the little program example shown above and change line 10 to read: 10 DIM A$(10), B$(20). Also change line 40 to substitute a comma for the semicolon after PRINT #1. RUN the program and watch where B$ is printed on the screen!

Oh yes...one more thing: Congratulations! You have just stored a piece of data on a disk and retrieved it! Not a program, real, honest to goodness data! You'll do much more of this in Chapter . But this is a historic first!

SAVE and LOAD

Format: lineno SAVE "filespec"

lineno LOAD "filespec"

Program Example: SAVE "D1:EXAMPLE1.BAS" (See Fig. 3.2 for filespecs)

This is just like CSAVE and CLOAD with only one exception: Filespecs are used to identify which program on diskette is to be SAVEd or LOADed. The filespec also tells the computer which drive to use if more than one is connected.

Note that SAVE and LOAD are I/O operations, but they are done in a "shorthand" form, and include the OPEN and CLOSE statements, which need not be used. All required procedures are included in the SAVE (output to disk) and LOAD (input from diskette to the computer) statements.

explain or include in def. in Append

Note too, that SAVE and LOAD deal with the program in tokenized form. Source programs may be stored character for character on diskette using LIST and ENTER which are discussed below.

As usual, SAVE and LOAD may be used in either direct or deferred programming modes. The former is more common, but occasionally you may wish to "chain" several programs together to RUN a game or whatever that is simply too large to fit all at once into available RAM. To do this, use LOAD at the end of the first part of the program in deferred mode, so that when the program encounters the LOAD statement, it will then read in the next part of the program from diskette. You must be careful, of course, that the new portion of the program stands alone, for all previous contents of RAM will be lost when the new LOAD is executed! (As you can see, program chaining is seldom a practical procedure, and it is generally better to purchase additional plug-in RAM cartridges as you need them).

LIST and ENTER

Format: lineno LIST _____ "filespec"

lineno ENTER _____ "filespec"

Program Example: 100 SAVE "D1: MARTY.VAR"

As you might have guessed from looking at the format and program example above, LIST and ENTER are used and formatted exactly the same as are SAVE and LOAD. LIST is the output (to diskette) command, while ENTER is the input statement. As with SAVE and LOAD, no additional statements are needed, for OPEN and CLOSE are included in these simplified I/O operations.

But why have two statements that do exactly the same thing? Well, they really don't, although the distinction between LIST and SAVE or between LOAD and ENTER may be more subtle than the beginning programmer really needs to know about. Put in the simplest possible terms, when we type a word or a statement on the keyboard and then press the RETURN key, we are ENTERing something. That "something" is called the "source" data. Source data is interpreted (during the RUN) by certain computer hardware and software into symbols that are useable ("executable" in computerese) by the machine. This process of interpreting the source into usable computer code is called "tokenizing" for the result is stored in RAM in the form of "tokens" which are meaningless to us poor Humans. When we SAVE a program, and then LOAD it back into RAM, we do so using the "tokenized" form of the program. LIST and ENTER do exactly the same thing, but in untokenized or source form! This takes considerably more memory, but there are times when we need to examine the source code used in a program, so it's a nice feature to have and to know how to use when the need arises!

DOS

Format: DOS

Places the system under the control of the disk operating system and causes the MENU (see Chapter 6)

To be displayed. IF no disk drive is connected, or no diskette is "booted", places the computer in "Memo Pad" mode.

CHAPTER 4: SEQUENTIAL ACCESS DISK FILES

CHAPTER 4: SEQUENTIAL ACCESS DISK FILES

As we've discussed (but briefly) you can do much more with an ATARI disk-equipped computer system than just store programs and RUN them over and over again. You can "chain" programs together, so that the last line in one BASIC program tells the disk system to RUN another program. But we've barely touched upon the real power of the diskette and disk drive: Data storage. This Chapter and Chapter 5 will attempt to remedy this gaping "hole" in the new disk user's education. We'll fill you up with some kind of information! OK? You're the computer, and I'm the diskette full of data. Get ready to access data! I can put it out all day...but if you aren't ready to READ me, no communication will take place...

By way of review, let's remember that data is just a computerese word for information. All data within the computer is stored and manipulated in the form of numbers. There are internal and ATASCII number codes for each character on the keyboard, each memory location has a number, and the instructions that the microprocessor can execute are also written as numbers. The computer feeds on numbers. They are its reason for being.

When we input or output data, or move it around in and out of the various parts of any computer system, it is important to know that the place where the data "lives" originally does not pass it along to another place! The data never leaves the original "residence". What really takes place is that the source location or device "shares" the data with the destination location (where data is needed) so that the destination can run out to the "office copier" and make a copy for its own use. Now the data is stored in two places! This is a very important concept to grasp. If we assign the value of 100 to the variable A, PUT the value of A (still 100, remember) into a disk file, GET the value back from disk and store it in the variable B, the same data (100) is now in three different places. It's stored in A, stored in B, and stored in the disk file as well!

Here is the program you need to do just what we have described:

```
10 GR.0
20 A=100
30 OPEN #3,8,0,"D1:DATAFILE"
40 PUT #3, A
50 CLOSE #3
60 OPEN #1,4,0,"D1:DATAFILE"
70 GET #1, B
80 CLOSE #1
90 PRINT "A= ";A
100 PRINT "B= ";B
```

We should remember that each time the file (D1:DATAFILE) is OPENed for OUTPUT, the end of file pointer is reset to the beginning of the file, which effectively erases everything stored in the particular file. Try running the program once to get the file established on disk, and then tell your warm-hearted little wonder to "GOTO 60" in direct mode. This should prove to your satisfaction that the data (100) is really stored on the diskette!

Some programmers and writers like to speak of "copying" data rather than "inputting" or "outputting" it. There is much logic in this approach. We will use all three terms in this book, as you will hear and read them elsewhere, and we want to familiarize you with as much of the "jargon" as we can. But remember how the process really works: Like the office copy machine!

4.2 Accessing your Data on Diskette

You've already run into the term "file" which is really a kind of a catch-all word for any collection of information manipulated by the computer, (generally in an I/O manner). To get to the bottom line, though, there are really only two kinds of files: A program file contains the instructions to the computer

that it needs to execute a program. A data file is a place to store information; whether the data stored is further manipulated by the computer or simply PRINTed out to the user.

Think of a file as a drawer in a common office filing cabinet. Each drawer can be locked, unlocked, opened, closed, and its contents aan be increased or decreased. If you type DOS you will see that these operations are considered routine diskette file actions, too, so the analogy with the file cabinet drawer really works. We can add something to the file with PUT# and PRINT# (note the # preceeds the "drawer number" to be used for the file operation). We can take things out of the file, run copies, and then replace them with GET# and INPUT#. When we SAVE a program on diskette, we are using one whole drawer to store the program. It is the filespec (such as D1:DISKFILE) that defines which "drawer" is to be used. The OPEN statement actually opens the drawer and allows us access to its contents.

So we have program files and data files. We've already discussed SAVE (create a program file), so let's take a closer look at data files. These files are distinctly split into two types, based on the method used to access their contents. Let's go back to our file drawer analogy for a moment: We can open one drawer and then look at the contents one at a time (computers can only do one thing at a time: They aren't really very smart, (just very very fast!) We look at each record stored in the file in the same order that it was originally stored there. In other words, if all the file folders were numbered in order of storage, we would first look at folder #1, then #2, etc. until we came to the folder or divider labeled "End of file". This would be an example of sequential access within the particular file drawer.

Of course, we have another option: We could open the drawer and then (using the numbers on the folders) pick out just the folder we wanted to examine. This is the other type of file access, and we will call this kind random access.

To a human file clerk, random access is just as fast and convenient as sequential access, as we would automatically pick the method best suited to the job at hand. For example, if the boss told our hypothetical clerk to "Get me folder #34", the random access method would obviously be faster and easier. We wouldn't even have to consider which method to use: We'd just get file 34 and give it to the boss. On the other hand, if the local VP in charge of filing told us "Get me all the files with numbers that end in 5, 7, or 9", the story would be different. We'd probably start with file folder number 1, then look at 2, then 3, and so forth, picking out only those that met the Veep's specifications. The sequential access method would probably be used.

Actually, when we talk about "random access" we don't mean "look at a file at random...you pick which one"! Not at all! We are <u>really</u> saying <u>direct</u> access to a particular piece of data in a file, without having to look at the numbered folders in order until we get there. Obviously, if there are a lot of folders in the drawer, random access is a far faster and more efficient procedure. But there are times when sequential access is preferable...

So let's take a good look, with example programs, of how the two methods work, starting with Sequential Access.

4.3 <u>Sequential Access Files</u>:

There are two methods of setting up sequential access files. If simple, one byte data entries are to be filed (such as the ATASCII codes or numerical data up to decimal 255), the easiest way is to use PUT and GET. We must remember that the end of file marker is moved back to the beginning of the file each time it is opened. For this reason, the most logical approach is to do our output to the disk file first, and then do any read operations from diskette. In this manner, there will always be data in the file when we attempt to OPEN it for input to the computer. If this were not the case, the computer would try to OPEN a non-existent file, and an error message and interrupted program execution

would result. Unfortunately, if a file does already exist, and we OPEN it for output to the disk, the pointer indicating the end of the file will be reset to the beginning of the file and all data currently in the file will be lost. "If we had any ham we could have ham and eggs if we had any eggs..." The net result of all this is that two facts must be kept in mind:

1. If we open an existing file for output (write to file), previous data will be lost.

2. If we open a non-esixstent file for input, an error will result. Of course, if we open such a "non-file" for output, the file will be created by DOS for us.

We'll spend more time later solving these two apparent problems, but for now, let's work our way through the simplest possible example of how we can PUT and GET data (byte by byte) to and from our diskette file. All we have to remember is that we must do a write to the file before we can expect to read anything, and that any time we write to the file, previous data will be replaced by what we have stored in the new write operation.

Spend a few minutes studying the flowchart of Figure 4.1 which shows our simple example PUT and GET operation program, and have a look at the program listing in Figure 4.2

Got that? Does it all make sense? Well, just in case, let's go through the program together...

After the usual introductory steps (title, DIMs, etc.) of lines 10-30, we get down to the nitty gritty in line 40 which asks the user whether a READ or WRITE operation is contemplated. This is indicated on the flowchart by the topmost "decision" box labeled "IS IT A "READ"? If it is not a read, it must be a write, and line 50 "falls through" to line 60 which "traps" user errors

FIGURE 4.1

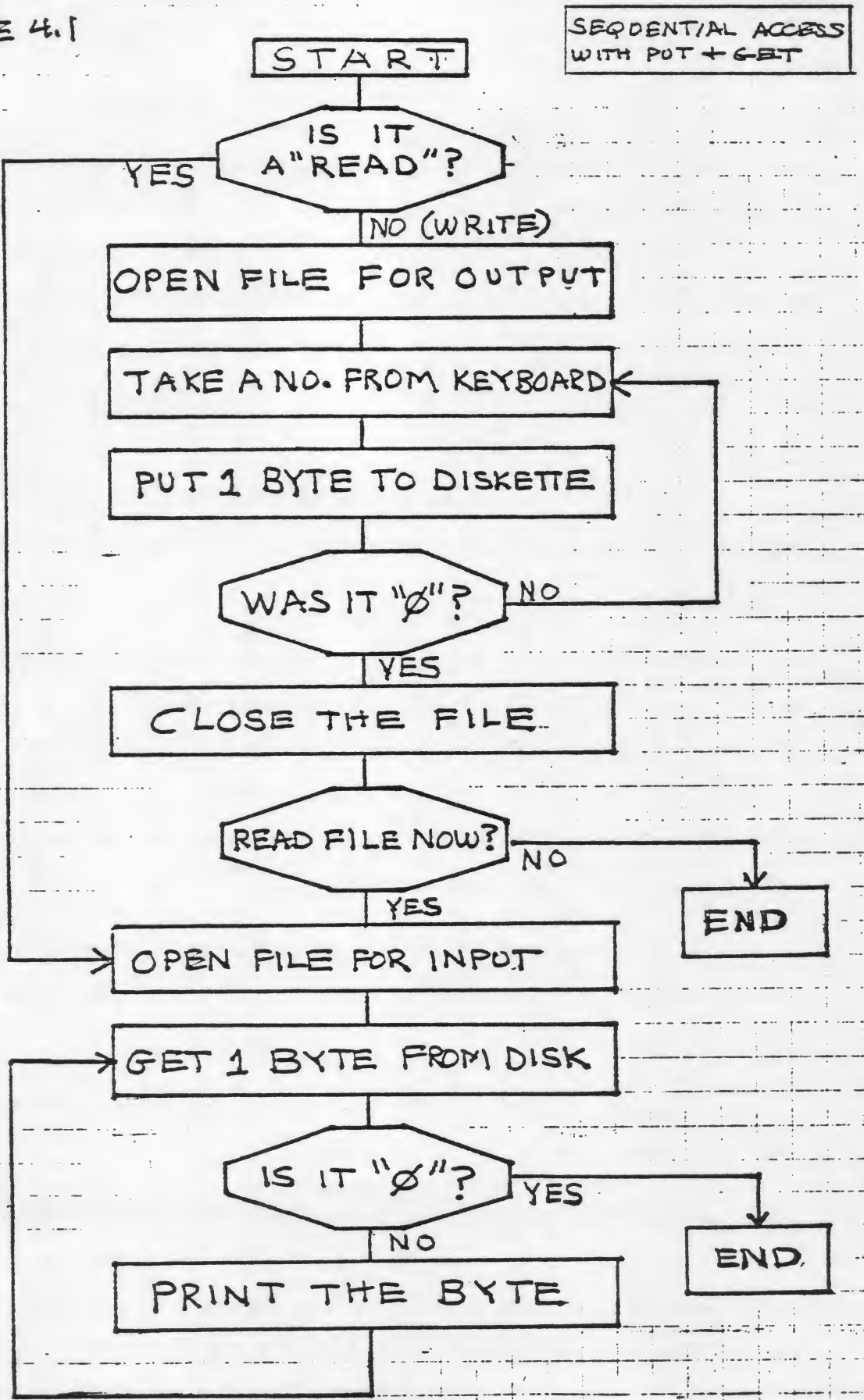

SEQUENTIAL ACCESS WITH PUT + GET
START
IS IT A "READ"?
YES
NO (WRITE)
OPEN FILE FOR OUTPUT
TAKE A NO. FROM KEYBOARD
PUT 1 BYTE TO DISKETTE
WAS IT "Ø"?
NO
YES
CLOSE THE FILE
READ FILE NOW?
NO
YES
END
OPEN FILE FOR INPUT
GET 1 BYTE FROM DISK
IS IT "Ø"?
YES
NO
END
PRINT THE BYTE

FIGURE 4.2

```
10 GRAPHICS 0:REM INTRODUCTORY MATERIAL
20 DIM A(50),A$(10)
30 GRAPHICS 0:? " PUT AND GET TO DISK PR
OGRAM EXAMPLE":?
40 ? "Is this to be a READ or a WRITE?"
;:INPUT A$:?
50 IF A$="READ" THEN 100
60 IF A$<>"WRITE" THEN PRINT :GOTO 40
70 REM WRITE ROUTINE
80 OPEN #1,8,0,"D:EXAMPLE1.DAT"
90 PRINT "Enter a number less than 256"
INPUT V
100 PUT #1,V
110 IF V=0 THEN CLOSE #1:GOTO 130
120 GOTO 90
130 GRAPHICS 0:? :? "Read data in file:n
o";:INPUT A$:?
140 IF A$="NO" THEN END
150 IF A$<>"YES" THEN 130
160 REM READ OUT ROUTINE
170 OPEN #2,4,0,"D:EXAMPLE1.DAT"
180 FOR E=1 TO 50
190 GET #2,G:A(E)=G
200 IF G=0 THEN GOTO 230
210 PRINT "BYTE # ";E;"=";G
220 NEXT E
230 CLOSE #2:? :? "       All Done, Mast
er!"
```

such as some wiseguy typing "Go Jump in the Lake, Buster!" In this case, line 60 evaluates the smart alack's answer as not equal to "WRITE" and sends the program back to line 40, where the READ/WRITE question is asked again. If the user has answered the question with "WRITE" then line 60 also "falls through" and the program passes on to line 70 (a REM statement), and then to 80, where the data file is opened for output (write). Reference #1 has now been assigned to this particular file. Line 90 asks the user to type in a number less than 255 (the highest that can be stored in one byte, hexadecimal FF). The user's number is assigned to the variable X. Line 100 PUTs that byte out to the disk file. Line 110 tells the program that if a zero is entered by the user, the output job is done, and the program then jumps down to line 130. If X is not 0, line 120 sends the program back to the output routine (but after the OPEN statement...to OPEN the same file more than once would be an error). Now another entry is taken from the user, PUT to the disk file, etc. until the user enters a 0. Now the program branches to line 130, which clears the screen and asks the user whether he wants to see the data that has been stored. If the answer is no (line 140) the program ENDs. If it is yes, line 150 "falls through", and we are now at 160, the REM statement, and finally end up at 170 which OPENs our file for input of data to the computer. Note that line 130 has CLOSEd the file for input. If this statement were not executed, we would get an error at line 170. Line 180 starts a FOR/NEXT loop that will be executed 50 times. In the loop, line 190 GETs one byte from the diskette file and assigns it to the array variable A(E).

In line 200, the byte received as a result of the GET statement (variable G or A(E)), is checked to see if it is 0. Remember, the last entry in the file was a 0. If it is a 0, the program goes to 230 and ends with the concluding printout "All Done, Master!" (you are its master, aren't you?) If G is not zero, the value of G is printed out (line 210), and the program loops back

to line 180. Note than when G is 0, line 230 closes the file.

This little program should be RUN as many times as necessary to give you a "gut feel" for what is happening in sequential access files. These steps are followed:

1. Decide whether a READ or a WRITE is to be accomplished.
2. OPEN the appropriate file.
3. Conduct the desired I/O operations, and do any necessary keyboarding of data or printing out of old data from disk.
4. CLOSE the file that was used for the I/O operations.

We must still overcome the minor problems involving the error caused by trying to open a file with no data (nonexistent file), and the problem of updating the data stored in a file without losing what's already there. These two items will be covered in the next section. Meanwhile, if you are a married madn, we strongly suggest that you do not demonstrate this first example program to your wife. Her reply is predictable: "You're nuts, Honey!" Our comment: She may well be right! To find the cure, read on...

OK, let's make a couple changes and see if we can't accomplish these things:

1. Make the file "updatable" by adding new data.
2. Make it do something practical: Like store your commonly called telephone numbers.
3. Make the program capable of handling more than one byte at a time.

To accomplish objective number 1, we can input the whole diskette file into RAM (if we don't have enough RAM, we can do this a piece at a time), store it as part of a very long string, add new data or subtract old obsolete stuff from the string, and finally re-record the whole string back onto disk. This solves

our problem of losing data every time we OPEN the file for output!

Objective number 2 can be accomplished: Just remember that your computer can do anything your mind can imagine! The limitation is between your own ears.

Finally, objective number 3 must be satisfied if we are to generate a practical phone directory program. We can do the job with PRINT# and INPUT# (the # is not part of the BASIC keywords, but is used to distinguish between these disk commands and the more mundabe usages of the same keywords).

Program planning always starts from the top and works down! This means that we must first decide the overall objective of the program, and then proceed with more and more detailed breakdowns of this one objective until at the very bottom of the pyramid the "code" itself can be generated. The objective of this program will be to teach you how to write a program for sequential access disk filing and retrieval of data using PRINT# and INPUT#, and coincidently to give you a "practical" program for storing frequently used telephone numbers. The numbers will be printed out in the form of a list on the monitor screen.

A total of two files will be needed for this program. The first is the disk file that contains the program itself. Let's call this one "D1:PHONLIST.BAS" (the BAS extender means that it's a BASIC program). The data file then can be called "D1:PHONLIST.DAT", with the DAT extender standing for "data". Since no numbers need be manipulated arithmetically (the telephone number has no value as a number), <u>all</u> data can be stored as part of a string. We can then input the entire string from disk, read certain parts of the string, even put your PHONLIST in alphabetical order! Once the data in the string has been "massaged" (silicon Gulch Computerese for "manipulated" or "processed"), and the appropriate print out to the user has been accomplished, the entire string can be output to the diskette file in one I/O operation using PRINT#.

At this point we have a fair idea of what we want our program to do. We can now break the program up into sections that computerists like to call "modules". This modular approach will make the program far easier to "debug" (render error-free) later, is easier to follow when read by another programmer (even you, when you read it over a couple years from now), and will be much easier to "code" (translate into executable computer language, BASIC in ATARI's case). To separate them from BASIC linenos, Roman numerals are used to identify modules within the program:

MODULE NUMBER	DESCRIPTION OF STEPS WITHIN THIS MODULE
I	Clear the screen, print out the name of the program, document with REM statements, DIM strings and matrices as required. This "scutwork" is also called "overhead" or "housekeeping".
II	Input the string stored on diskette and store it as A$ Close the disk file.
III	Prompt message to user: Do you want to look up a number or add a number? If user wants to add data, send program down to module IV. If not, module V.
IV.	Take input from user for new names & numbers and store by tacking on to end of A$. Send program to subroutine to alphabetize entries. (See alpha-sort in Appendix A of BASIC Reference Manual.)
V.	Print-out. Substring by substring, print out the names and numbers in the storage string (A$) on the screen. When done, ask user if data correct. If yes, then go on to module VI.
VI.	PRINT# A$ to diskette. First OPEN the file for output, do the PRINT#, then CLOSE the file. Display "end" message to user.

The next step in the creation of our program is to write the "code" that will cause each of our modules to be properly executed and then pass the result along to the next (or other applicable) module for further processing. The code for the phone directory program appears as Figure 4.3. We feel that the best way to "write code" is so sit down in a comfortable chair at a desk or table and write each BASIC statement in longhand. A REM statement "leads off" each module, specifying what the module number is and giving a very brief summary of what that module does. Roman numerals may be used for numbering the modules, as we have done in this program. Here's an example of the type of REM statement we are talking about:

```
100 REM MOD II: TAKE INPUT, STORE IN MATRIX A
```

In this case we are told that line 100 is the first line in module II, and that the module will prompt the user for input, take the input, and assign it to the various matrix elements in matrix A.

We hope you see some of the advantages of the modular approach (and the top down design) in programming! The final step in writing a program will really drive home the advantages of such a logical approach:

The last step in programming involves entering the program line by line into the computer, and making trial RUNs. But don't forget you're a disk user now, so be sure to 'SAVE your program before you RUN it for the first time! A program that RUNs perfectly the first try is a once in a lifetime thing. If this happens to you, the fatted calves, dancing girls, and iced champagne should be trotted out, and a week of festivities should be initiated at once

It is more likely that a period of "debugging" will ensue, so just accept that as the fate that befalls all programmers, amateurs and professionals alike! Good (bug) hunting!

EXAMPLE2.BAS

PHONLIST.DAT

A PRACTICAL PHONE DIRECTORY PROGRAM

```
10 REM MOD I: HOUSEKEEPING. 40 ENTRIES WI
LL FIT IN 1000 BYTES
20 DIM A$(1000),D$(20),D$(50),B$(5):A$="
"
30 GRAPHICS 0:? :? "       TELEPHONE DIR
ECTORY":? :?
40 D$=" "
100 REM MOD II: INPUT FILE FROM DISK TO
A$
110 TRAP 200
120 OPEN #1,4,0,"D:PHONLIST.DAT"
130 INPUT #1;A$:TRAP 40000:CLOSE #1
200 REM MOD III: QUERY USER LOOK-UP NUMB
ERS OR WRITE NEW DATA?
210 ? "Do you wish to see numbers (S) or
 add new numbers (A)?":?
220 INPUT B$:IF B$="S" THEN 400
230 IF B$<>"A" THEN 210
300 REM MOD IV: TAKE NEW USER DATA, ALPH
A-SORT WHOLE LIST
310 GRAPHICS 0:? :? "Please enter last n
ame 1st initial and phone number: (Smith
 J 222-2222)":?
320 INPUT D$
330 IF D$="Q" THEN 390
340 IF LEN(D$)<25 THEN D$(LEN(D$)+1)=D$:
GOTO 340
350 A$(LEN(A$)+1)=D$:GOTO 310
360 GOSUB 2000
370 GOTO 600
400 REM MOD V: PRINT OUT WHOLE STRING ON
 SCREEN DIVIDED WITH 25 BYTE SUBSTRINGS
405 GRAPHICS 0:POKE 82,7:PRINT
410 FOR A=25 TO LEN(A$)-25 STEP 25
420 PRINT A$(A,A+24)
440 NEXT A
450 POKE 82,2:END
500 REM MOD VI: WRITE STRING TO DISK FIL
E
510 OPEN #2,8,0,"D:PHONLIST.DAT"
520 PRINT #2;A$
530 CLOSE #2
540 GRAPHICS 2:SETCOLOR 2,0,0:COLOR 1
550 PRINT #6;"      Thank You!"
1000 END
2000 REM ALPHA-SORT ROUTINE
2005 IF A$(1,25)="000000000000000000000000
000" THEN 2020
2010 D$="000000000000000000000000000000777777
777777777777777777"
2020 A$(LEN(A$)+1)=D$
2030 L=LEN(A$)-25
2040 FOR I=25 TO 25*L STEP 25
2050 FOR L=25 TO 25*L STEP 25
2060 IF A$(L,L+24)<A$(L-25,L) THEN GOSUB
 2080
2070 NEXT L:NEXT I
2080 RETURN
2090 END
2100 REM SWITCH ENTRIES
2110 D$=A$(L,L+24)
2120 A$(L,L+24)=A$(L-25,L)
2130 A$(L-25,L)=D$:RETURN
```

CHAPTER 5: RANDOM ACCESS DISK FILES

CHAPTER 5 - RANDOM OR "INDEXED" ACCESS DISK FILES

5.1 Introduction:

We've seen how sequential access disk operations are conducted. In many cases, for the sake of speed of execution, we start by bringing everything into RAM, and then do our sorting, searching, and other "massage" of the data electronically. This, of course, is quite fast, and is adequate for most personal computer needs. But if we are dealing with a fairly large mass of data, we may very easily run out of RAM with these techniques. To get fast return of information from disk without the RAM limitations, we must use a different approach, and this is usually called "Random access" disk operation. ~~We call it that~~ This is because any piece of data we need can be pulled in from disk without first "~~wading~~ wadering through" other data and rejecting it, (a very slow process which partially defeats the intended purpose of the disk hardware!) Actually, as we will see, the access to the desired data is not random at all, but could better be termed "direct", "directed", or "indexed" access. Try to bear all this in mind as we wade in a little deeper...

5.2 The ~~"Direct" or~~ "Directed" Access Concept: Obviously, if the access to data stored on a diskette were truly random, we would have no way to predict or control what data would be input to the computer. In turn, the poor computer would probably become terribly confused and yield only "garbage" for output or (more likely) print out a message casting aspersions upon the programmer's time, place, and method of origin!

The answer, of course, is for us to work out a practical way to tell the computer (and through it, the disk drive) exactly where the data we want is physically located on the disk. With this kind of information, the computer can "set" the drive pick-up head to the location (sector) of the diskette where the desired data is located. If you remember the descriptions of NOTE and POINT in Chapter 3, you'll have a pretty good idea of how we do this...

If we type a message, number, or whatever into the computer and store it in the variable A$, we can PRINT# that string onto the disk. But if we first do a NOTE#, and store the sector and byte where the string will be stored (~~really~~ actually, the first character will be stored there) in two numeric variables like A and B, we could then go back and get the data we stored by first doing a POINT# ,A,B! Let's take a look at a simple directed access disk program and see how this might look reduced to the very simplest possible form:

Type this into your computer and store it on diskette using SAVE:

```
10 GR.0: PRINT
20 DIM  TEXT$(128), READ$(128)          (Message may be up to 128 bytes long)
30 OPEN #1,8,0,"D:DIRECTAC.DAT"         (Open the data file)
40 PRINT "TYPE YOUR MESSAGE UP TO
128 CHARACTERS"
50 INPUT TEXT$: PRINT: PRINT
60 NOTE #1,. SECTOR,BYTE                (Store starting sector and byte)
70 PRINT #1; TEXT$                      (Note the semicolon!)
80 CLOSE #1: PRINT "YOUR TEXT NOW
STORED ON DISKETTE"
90 FOR N=1 TO 2000:NEXT N: GR.0         (Time delay, clear the screen)
100 REM PRINT OUT ROUTINE
110 OPEN #1, 4, 0, "D:DIRECTAC.DAT"     (Open the file for input to computer)
120 POINT #1, SECTOR, BYTE              (Set up disk head for correct data)
130 INPUT #1, READ$                     (Bring in data and store in READ$)
140 CLOSE #1
150 PRINT READ$
160 FOR N=1 TO 2000:NEXT N
170 ?:?:?"GEE WHIZ I'M A SMART          (Yes!  And modest, too!)
COMPUTER!"
```

NOT RUN - No equal!

When you have RUN this program a few times you will see that the message you put in as TEXT$ is stored on the disk, then brought back into RAM as READ$. This is so you can have it (direct mode) PRINT TEXT$ and then PRINT READ$ and compare what you typed in with what it gave you back from the diskette. Here's another hint. With your disk in place, try typing DOS to get the menu, and notice that item C is COPY. Type ~~item~~ C, and you will get the message "COPY FROM,TO". Answer this by telling it to copy from your diskette data file to the screen editor device. Oh! Don't remember how to do that, huh? Well, just type DIRECTAC.DAT, E: Now press RETURN and watch the screen. The computer will go out to the disk, bring in everything that is in the file, and print it out on the screen for you! Remember this prodedure! It's a really easy way to determine whether your program"s bugs are in the write routine or the read portion...

When you display the data in the file on the screen in the above manner, you'll notice that only one item is displayed, even though you may have run your program several times. This is proof that when you OPEN for output (to store something), what is already "out there" on diskette is lost and gone forever. The other drawback to our "stripped down" indexed access example program is that the sector and byte (NOTE and POINT) data is lost (not stored on diskette.)

Now let's see if we can't solve these problems with a really practical program that will let us write memos that can be read later to remind us of what we had planned for a particular day in the future! Again, we'll use Top Down (it's not raining, is it?) Design and modular programming techniques. ~~We'll start by writing a paragraph that will describe just what we want our program to do:~~

5.3 A "Practical" Program Using Indexed Disk Access

Objectives: This program works like a desk calendar. For any future date, a message (up to 128 characters long) can be stored on disk and then recalled by date. Note that even a real, paper-type desk calendar is somewhat "sequential" in operation: You flip to a date, see whether your "wanted" date is ahead or behind that date, then page through (sequentially) until you come to the desired date. Our program, on the other hand, goes directly to the particular date using NOTE and POINT to "set" the disk drive head to the position of the desired data!

Modular Programming: Our next programming task, now that we know what we want the program to do, is to divide the total job into "bite sized" pieces or modules that are convenient for the programmer to handle. These modules are never for the purpose of making the computer's job easier! They simply ease the programmer's complex tasks. Let's start with Module I:

Module I Input

Clear the screen, get in correct graphics mode, DIM all strings and matrices. Accomplish any other "housekeeping" needed for the program. Finally, bring into RAM from disk the entire index file (list of NOTE and POINT data, stored by date). Store this in an array

Module II Input

Using a loop, prompt the user to enter the date and then the text of his memo. If he enters a "0" for the date, jump out of the loop and go to module IV, which prints the index matrix back out to disk in its up-dated form (including the material entered in this module). Note that we are talking about only the index matrix, not the memo string itself. This will go in a second data file.

Module III Input

While still in the loop started in Module II, first do a NOTE to get the starting location of the string location on disk, then PRINT# the memo as a string to diskette. Go back to module II for the next entry. Note that the only exit from this module is back to module II! Note also that this module contains the only routine for PRINTing new memos onto diskette.

Module IV Input

PRINT# the entire index matrix to diskette. CLOSE all files. Note that data may be lost if the files are not CLOSEd before the end of the program!

Module V Input

Ask user whether he wants to read any memos now. If his answer is yes, RUN the output routine "D:MEMOGET.BAS". If his answer is no, then end the program with an appropriate message (such as "thank you!", "It's always a pleasure to serve you!" or some-such corn).

At this point, you have really "written" the program! All that remains now is to code it in terms the computer can execute. Before we start punching keys, though, let's write the output program.

Module I Output

DIM, Clear the screen, choose graphics mode, title, other housekeeping as usual.

Module II Output

Bring in index data from disk and store in matrix. First get date, and if it is 0 then stop. If not 0 then bring in sector and byte data for POINT.

Module III Output

Prompt user to enter date. Compare the date entered with the various dates in the index matrix. If a match is found, then then set matrix variables for sector and byte equal to the variables to be used in the POINT statement. If no match is found, print a message to that effect.

Module IV Output

Open the file that contains the actual memo texts, and using POINT with the variables established in module III, retrieve the particular memo for the requested date. When memo has been brought into ram as A$, close the files.

Module V Output

Clear the screen by going to GR.0, then print the date of the memo and its text. Skip a couple lines, then ask the user if there are more memos he wants to read. If yes then go back to module III. If no, then prompt user by asking whether he wants to write any new memos. If he does (answers Yes) then program chains to the write program (MEMOPUT.BAS) using RUN "D:MEMOPUT.BAS". If the user answers no, then end the program with an appropaiate message.

The coded program is shown next. Note the TRAP statements and the "subroutine" (line 1000) that is called by the TRAP on line 30. The line 50 TRAP 40000 statement is especially interesting. Here's what we're doing in Module I of the write memo routine:

Line 30 TRAPs the OPEN for input and output statement. If there is an error, line 1000 will be executed. If not, the other file will be re-OPENed on line 40, and the TRAP 40000 statement will be encounteded on line 50. Since the highest permissible line number in ATARI BASIC is 32676, TRAP 40000 in and of itself is an error, and the TRAPS will be "sprung", thus effectively

EXAMPLE OF INDEXED ACCESS

INPUT ROUTINE

```
REM MODULE I:HOUSEKEEPING. BRING INDE
X MATRIX A IN FROM DISK, GET FILES OPEN
  DIM A$(128),A(300,3),YN$(10):GRAPHICS
 0
30 TRAP 1000:OPEN #1,9,0,"D:MEMOPUT.DAT"
40 OPEN #2,9,0,"D:MEMOPUT.IND"
50 TRAP 40000
60 FOR R=1 TO 300:A(R,1)=0:A(R,2)=0:A(R,
3)=0:NEXT R:TRAP 110
70 FOR R=1 TO 300
80 INPUT #2;I:A(R,1)=I:IF A(R,1)=0 THEN
POP :GOTO 120
90 INPUT #2;I:A(R,2)=I
100 INPUT #2;I:A(R,3)=I:NEXT R
110 T=R
120 TRAP 40000
200 REM MODULE II:GET MEMO & DATE FROM U
SER
210 PRINT "        MEMO FILE: WRITE PROG
RAM"
220 FOR R=T TO 300
230 ? :? "Memo for what (DDMMYY) Date?":
?
240 INPUT DATE:IF DATE=0 THEN 300
  ) A(R,1)=DATE
260 NOTE #1,X,Y:A(R,2)=X:A(R,3)=Y
270 PRINT :PRINT "Text of Memo (Up to 12
8 Characters...":?
280 INPUT A$:PRINT #1;A$
290 NEXT R
300 CLOSE #1
400 REM MODULE IV: PRINT MATRIX A TO DISK
410 FOR R=1 TO 300:IF A(R,1)=0 THEN 440
420 PRINT #2;A(R,1):PRINT #2;A(R,2):PRIN
T #2;A(R,3)
430 NEXT R
440 CLOSE #2
500 REM MODULE 5:CHAIN TO READ PROGRAM?
510 GRAPHICS 0:? :? "DO YOU WANT TO READ
 MEMOS NOW?":?
520 INPUT YN$:IF YN$="YES" THEN RUN "D:M
EMOGET.BAS"
530 IF YN$<>"NO" THEN 510
540 IF YN$="NO" THEN GRAPHICS 0:PRINT :P
RINT "                Thank You."
 9 END
1000 CLOSE #1:OPEN #1,8,0,"D:MEMOPUT.DAT"
1010 OPEN #2,8,0,"D:MEMOPUT.IND"
1020 CLOSE #1:CLOSE #2
1030 GOTO 30
```

READ-OUT ROUTINE

```
10 REM MODULE I:HOUSEKEEPING
20 DIM A$(128),A(301,3),YN$(10)
30 GRAPHICS 0:? :? "          MEMO READ
PROGRAM":? :?
100 REM MODULE II: GET INDEX MATRIX INTO
 RAM
110 OPEN #2,4,0,"D:MEMOPUT.IND"
120 TRAP 200
130 FOR R=1 TO 300
140 INPUT #2,I:A(R,1)=I:IF A(R,1)=0 THEN
 POP :GOTO 200
 POP :GOTO 200
150 INPUT #2,I:A(R,2)=I:INPUT #2,I:A(R,3)=I
160 NEXT R:CLOSE #2:TRAP 40000
200 REM MODULE III: GET DATE AND LOOK UP
 IN INDEX MATRIX
210 PRINT "Please enter date of memo to
be looked up"
220 TRAP 220:INPUT DATE:TRAP 40000
230 ? :FOR R=1 TO 300
240 IF A(R,1)=0 THEN POP :PRINT "Sorry!
No memo in File for ";DATE:GOTO 210
250 IF A(R,1)=DATE THEN POP :GOTO 270
260 NEXT R
270 SECTOR=A(R,2):BYTE=A(R,3)
300 REM MODULE IV: GET MEMO FROM DISK
310 OPEN #1,4,0,"D:MEMOPUT.DAT"
320 POINT #1,SECTOR,BYTE
330 INPUT #1,A$
340 CLOSE #1
400 REM MODULE V:PRINT OUT MEMO
410 GRAPHICS 0:? :? :? "MEMO FOR ";DATE:
? :?
420 PRINT A$:?
500 REM MODULE VI: USER DECISIONS
510 PRINT "Any More Memos to Read?":INPU
T YN$
520 IF YN$="YES" THEN 210
530 IF YN$<>"NO" THEN 510
540 GRAPHICS 0:? :? :PRINT "Want to writ
e new Memo(s)?"
550 INPUT YN$:IF YN$="YES" THEN RUN "D:M
EMOPUT.BAS"
560 IF YN$<>"NO" THEN 540
570 GRAPHICS 0:? :? :? :? :? :? "
      THANK YOU!":END
```

removing them from the program! But why the TRAP statement at line 30 in the first place? Well, it's like this, folks...If we try to open a file for input (with code numbers 4 or 9), and that file doesn't yet exist (such as during the first execution of the program), an error will result, and it is that error that we are TRAPping. Once we get below the OPEN statements at lines 30 and 40, we'd like to get rid of the TRAP so we don't later TRAP another kind of error and send the program to the wrong place! (Note that in the read-out program, the TRAP on line 120 is for an entirely different purpose: Here we are TRAPping for a possible out-of-data error. But the TRAP 40000 "detrapping" statement is present on line 160).

5.4 Summary of Directed or Indexed Access Diskette Programs

We think that once you get this example program "punched up" on your system and RUN it a few times, you'll be amazed at the speed with which the disk operations are carried out. Access is almost (at least in human terms!) as fast as to data stored in RAM. But what is stored in your precious RAM is only the index matrix! The real data stays on disk and is brought in 128 bytes at a time.

You'll find Indexed Access diskette programs extremely useful any time file data must be stored with rapid retrieval and easy updating. The more you use the techniques discussed in this Chapter, the more uses you will find for them. Here's a brief outline of the steps that need to be followed to do practical indexed access file manipulation:

1. Get the two files opened for input (or input/output with 9)
2. Input the index data and store in a matrix consisting of your "key" number, the sector, and the byte numbers.
3. To output a new record onto disk, get the data from the keyboard and put in a string variable. Then do a NOTE and store the

tw o numbers (Sector and Byte) in your matrix, then PRINT# the string of text to diskette.

4. To input a particular record, get the key number from the user, look it up in the matrix to get the sector and byte numbers, then do a POINT# to "cue" the disk pick-up head. Now INPUT# the text string, storing it in a string variable for appropriate later use.
5. Close the text$ file.
6. Output the index matrix to the index disk file. Close the file.

Note that the NOTE and POINT reference numbers (immediately following the # sign) always refer to the string storage file, not the index file!

5.5 A final word or Two

When we use directed or indexed access routines, a matrix or some similar method is used to "POINT" the disk drive to the correct physical data location on the diskette. This matrix is sometimes called (in computer jargon) a "hashing table". In the example above, we used dates as "key" numbers in our primitive "hashing table". We could, of course, have dropped the two ending digits that indicate the year, taken the ATASCII values of an alphabetical string, or done virtually any other computerized calculation that seemed appropriate...Use that old imagination! Have fun!

CHAPTER 6

THE DOS MENU ITEMS AND HOW TO USE THEM

CHAPTER 6 - THE DOS MENU ITEMS AND HOW TO USE THEM

6.1 About the "Menu"

If you haven't already, you will eventually see that the "menu" approach is used quite a lot in computer programming. It allows the programmer to display a whole slew of different options that the user of the program may choose. In simple terms, it replaces what would otherwise be a large number of individual input statements and prompts of the "Do you want to Read or Write to the file?" variety.

A computer menu is very much like a restaurant menu. It displays to the customer the choices available to him at the time. The customer or user may then "order" the item of his choice, either by simply entering a letter or number, or by making such entry followed by use of the RETURN key. The DOS menu is a fine example of this type of programming, and should be studied both for its content and for the way in which the menu approach may be used in your own programs.

```
DISK OPERATING SYSTEM                          9/24/79
COPYRIGHT 1979 ATARI

A. DISK DIRECTORY                  I. FORMAT DISK
B. RUN CARTRIDGE                   J. DUPLICATE DISK
C. COPY FILE                       K. BINARY SAVE
D. DELETE FILE(S)                  L. BINARY LOAD
E. RENAME FILE                     M. RUN AT ADDRESS
F. LOCK FILE                       N. DEFINE DEVICE
G. UNLOCK FILE                     O. DUPLICATE FILE
H. WRITE DOS FILE

SELECT ITEM
```

6.2 The Menu Items and What They Can Do For You

Let's examine each item on our DOS menu and discuss its uses and how you can make it work. First, though, let's nobe once again that you don't have to type the full name of the item you want: Just type the letter next to the item (no period necessary), and hit RETURN. From that point, the DOS itself will prompt you on other information the computer needs to carry out your instructions.

Item A on the menu is the one labeled DISK DIRECTORY. The purpose of this selection is to let you see the name of each file on the particular diskette. If you type the letter A (with the menu displayed, of course), and hit RETURN, the computer will ask you to specify the file to be looked up. Usually you won't be able to answer this question, so just hit RETURN again, and all files on the diskette will be listed on the screen. When you are through with the list, hit RETURN once more to get back to the Menu display. Note that files are displayed in two columns under the Disk Directory selection. The first column is the file name, and the second is the extender. See Figure 6.2. The number following the filename and extender is the number of sectors currently used by the particular file. The number of available (unused) diskette sectors is also displayed (below the last file in the directory list)

Item B on the DOS Menu is RUN CARTRIDGE. This refers to the cartridge installed in the computer, normally BASIC. Thus, we can return to BASIC from DOS at any time just by typing the letter B and hitting RETURN onee. It should be noted here that if we have a program or parfial program in RAM, and we type DOS to make use of a Menu item, we may return to our BASIC program by use of the RUN CARTRIDGE command, and the contents of RAM will not be disturbed. (You can do the same thing by simply hitting the SYSTEM RESET key. Again, RAM will not be disturbed). In either case, you will get a cleared screen and the READY prompt. Type RUN to start program execution, or use LIST, etc. as desired.

Item C is the COPY FILE command. This simply lets you copy a file from the diskette to another diskette file, to the screen, printer etc. When you type C followed by RETURN, you'll get the prompt "COPY, FROM, TO". Answer by giving the filename of the source file, a comma, and then the filename or device code for the target file. The destination may be a totally new file which does not exist as yet. As an example, let's say you have an old file called "NUMBERS.BAS" and you want to put it in a new file to be named "MATHFILE.BAS". Here's how to proceed. Boot up the disk as usual, and type DOS. Now type C followed by RETURN. The screen will prompt you with COPY, FROM, TO. So tell it! Just type NUMBERS.BAS, MATHFILE.BAS and hit RETURN. Note that it is not necessary to use quotation marks or the designation D: to use the COPY FILE menu feature.

If you want to see what is in a file, you can use menu item C as follows: Type C and hit RETURN. Answer the prompt with FILENAME.BAS, E: (remember the E: is the device code for the screen editor device). The disk will input the contents of file FILENAME.BAS and the computer will output the information to the screen editor which will display it on your monibor. You'll find this an unusual and very useful feature!

Item D on the DOS Menu is DELETE FILE(S). This let's you get rid of files that you have copied onto other diskettes or for which you have no further use. Deleting these extrania provide you with more space for new material on the diskette. Typing D followed by RETURN will bring up a prompt asking you to enter the name(s) of the file(s) to be deleted. Once again, filename and extender only need be typed. Don't use quotation marks or the D: device code. When you get the filename(s) entered, press RETURN. Now the computer wants to verify that you really intend to delete the files, and that you have entered the right names. So it "reads back" the file name(s) you entered and tells you to "TYPE Y TO DELETE". To proceed, follow the prompts by typing a Y and then hitting RETURN. If you have mistakenly entered the wrong file name or have pushed D by mistake, simply press RETURN, and the computer will come back with its SELECT ITEM prompt.

Item E is RENAME FILE. Here you will be prompted to enter the old file name and the new one. The system will then change the name of your file as you have specified.

Item F is LOCK FILE. This means that a lock will be placed on the specified file so that it cannot be OPENed without first UNLOCKing it (item F). Once again, the system prompts are straightforward and easy to follow. When you type menu item F, the computer will ask you what file to lock, and you respond by typing in the file name and extender without the D: or quotation marks. If you are working with a data file, remember to lock the correct file! For example if FILENAME.BAS provides information to or gets it from FILENAME. DAT, AND FILENAME.BAS IS really only the program while the data we want to lock is in the file with the DAT extender, we want to LOCK DAT not BAS... Got DAT straight? Good!

Item G, UNLOCK FILE, is the command used to UNLOCK a previously LOCKed file. Going to our diskette directory by using menu item A (don't forget to press RETURN twice) will give us a list of all files on the diskette. Any that are LOCKed will have * just to the left of the filename! If we want to UNLOCK a file, we just type G, and the computer will ask us which file to UNLOCK. As usual enter the filename and extender without D: or quotation marks and press RETURN . When SELECT ITEM reappears, you know the job is done. You can check by typing A and RETURN and checking the list of files on the diskette. The asterisk next to the file your just UNLOCKed should now be gone.

Item H on the Menu, described as WRITE DOS FILE was mentioned in Chapter 2. Each time a new diskette is procured, it must be formatted and the DOS must be copied onto it. To use this item, we first Boot Up with an old diskette with the DOS on it. This gives us the menu display. We then have the contents of the DOS software in RAM, and we can remove the diskette and insert a new totally blank diskette in the drive. The command H is used only after formatting is completed using command I.

Item I is the command to FORMAT DISK. This delineates the various magnetic "tracks" on the new diskette prior to writing the DOS software onto the disk using command H. See Chapter 2 for a full description on formatting and copying the DOS file onto a blank diskette.

Item J lets you duplicate an entire disk. In computerese, we call the two disks the "source" and the "destination". Start with the source disk in your drive (drive 1 if you have more than one). If you have two or more drives connected to your system, put the desitnation diskette in drive 2. If you have only one drive, just keep your destination diskette close at hand for now. Press menu item J, and the computer will come back with

DUP DISK - SOURCE, DEST. DRIVES

If you are using one drive, enter 1,1. If you are using more than one, enter 1,2 (or 1,3 or 1,4). The source diskette should normally be placed in drive 1. Once this entry has been typed, with the two drive numbers separated by a comma, press RETURN. If you are using more than one drive, the duplication process will now take place automatically. If you have one drive connected, the computer will now prompt as follows:

INSERT SOURCE DISK, TYPE RETURN (so do it!)

After about 30 seconds of disk operation, beeps, clicks, etc., the screen will prompt you again:

INSERT DESTINATION DISK, TYPE RETURN (once again, do it!)

After another approx. 30 seconds, the computer will prompt with ... SELECT ITEM which means the duplication job is done and you can proceed to other tasks.

~~Item K on the menu is BINARY SAVE.~~

Item K on the DOS Menu is BINARY SAVE. This command is used to save on diskette a binary or machine language program. When this command is executed, the machine program must be in RAM, and when K is pressed, followed by RETURN, the computer will prompt you with the words

SAVE - - GIVE FILE, START, END

This indicates that you must provide the computer with three things. First type the filename and extender if used, followed by a comma. Second comes the starting address in hexadecimal notation, again followed by a comma. Finally, the hexadecimal ending address must be provided. Here are a couple of examples:

BINARY.P12, 31A, 33D

JEANINE.111, 20D3, 212D

Your machine-level program will be SAVEd on diskette as soon as you press RETURN. It may be reloaded by use of Item L (see below), BINARY LOAD. But remember that these commands are for use with machine language programs only, and cannot be used with BASIC commands or programs.

Item L. BINARY LOAD. When you wish to load a binary program SAVEd by use of Item K (see above), BINARY LOAD is used by entering L. The computer will respond by prompting you as follows:

LOAD FROM WHAT FILE?

You must then provide the filename to be loaded.

Item M. RUN AT ADDRESS. This is the command you will use to RUN your binary program. When you type M followed by RETURN, the computer will ask you

RUN FROM WHAT ADDRESS?

The requested parameter is the hexadecimal address to which the program control will be transferred (transfer address).

Item N is the DEFINE DEVICE command. It creates a logical (i.e. nonexistant or imaginary) device with the device name you specify. As you know, system created devices (printer, keyboard, diskette, screen, cassette program recorder, etc.) have their own device codes. This command lets you create your own device code for later reference. When you type the letter N followed by RETURN, the computer will give you the following prompt:

LOGICAL DEVICE, PHYSICAL DEVICE?

It wants you to tell it what the code letter (followed by a colon, as usual) will be for the new device, and then what the name of the device will be. For example, let's say you want to connect (through the normal I/O plug) a device for output, and you will call this device OUTPUT. Your prodedure would look like this:

- First you would type N followed by RETURN
- Next the computer will print LOGICAL DEVICE, PHYSICAL DEVICE?
- You would then respond with something like this:

  X:, OUTPUT (followed by RETURN)

From this time on, you can re fer to the defined device as X: (exactly as you would D: for a disk drive, P: for a printer, etc.) You can copy from a disk file to the defined device or conduct any other desired I/O operation using the device code defined (X: in our example...but any unused letter will do!)

Item O on the menu is DUPLICATE FILE. This refers to the duplication of a particular file on one disk onto a second disk. The procedure is just like Item J, except that since there is less to duplicate, less time is consumed in the process. Again, if you are using only one drive, you will be prompted when to insert the source and destination diskettes. The procedure goes like this:

8	(you type)
NAME OF FILE TO MOVE?	(computer queries)
FILENAME.BAS	(you answer it)
INSERT DESIINATION DISK, TYPE RETURN	(computer prompts you)
SELECT ITEM	(like READY in BASIC)

CHAPTER 7

PROGRAMMING HINTS AND IDEAS FOR ATARI DOS USERS

CHAPTER 7 - PROGRAMMING HINTS AND IDEAS FOR ATARI DOS USERS

7.1 Introduction

As seems to be the normal condition with computers, your uses for your Disk Drive are limited only by your own imagination! The personal computer and all of its peripherals are machines without function...until you tell them what to do! So let that old imagination take wing! Who knows what will fly out of your "magic box" next?

As a sort of a launching pad, this Chapter will cover some of the often used but seldom thought of techniques that you can "borrow" from other programmers. Program chaining will be discussed, as will a simple method of saving a whole bunch of RAM in cases where you won't need the DOS Menu items for the particular program. A few "practical" examples written by the ATARI software staff are included for your education and enlightenment...

7.2 Program Chaining

You may remember we used this concept in connection with the cassette recorder in our discussions in the BASIC Reference Manual. The grass roots idea of program chaining is to get around memory limitations (maybe you have only 16K of RAM installed in your computer, the DOS/FMS software eats up about 9K of that, and you find yourself with only 7K-bytes left for your 30K program).

If you can break your program up into five parts (often called "modules" by the pros), you can have the last line of each part RUN the next part by a simple statement such as

```
lineno RUN "D aexp : filename . (extender)
```

Here are a couple examples:

```
2340 RUN"D1:BALANCE.BAS"
1200 RUN"D:BALANCE.NO1"
```

Note that the closing quotation mark is optional, and that the aexp that follows D specifies which drive is to be used. If no drive is specified, drive 1 will be used by default. Therefore (as in our second example above) the drive number expression is optional.

This technique can be repeated as often as necessary. It is especially useful for "chaining" graphics programs (see the program called MOIRE-WAVER) where it is not necessary to carry over data, variable assignments, etc. from one part of the program to another. You should be especially wary of this fact: Each part of the program will be treated by the computer as a completely fresh, new program, and all computations previously accomplished and stored in RAM <u>will be lost</u>!

PROGRAM "CHAINING" EXAMPLE

MOIRE → WAVER

MOIRE.BAS

```
5 DIM OPTS(30)
10 GRAPHICS 8
20 COLOR 1
30 ROWS=200:COLS=192
40 XCENT=ROWS/2:YCENT=COLS/2
50 REM CIRCLE ROUTINE
60 FLAG=1
70 YS=0.9
90 FOR A=1 TO 30
100 OPTS(A)=SQR(400-A*A)
110 OPTS(A)=OPTS(A)*YS
120 NEXT A
130 R1=7:R2=200:R3=5:OF=5
135 IF R2>110 THEN R2=110
140 GRAPHICS 56
150 FLAG1=1
160 SETCOLOR 2,1,1
170 FOR R=R1 TO R2 STEP R3
180 XCENT=XCENT+FLAG1*OF
190 YCENT=YCENT+FLAG1*OF
200 SC=R/22
210 PLOT XCENT,YCENT+FLAG*R*YS
220 FOR A=1 TO 30
230 X=A*SC
240 Y=OPTS(A)*SC
250 DRAWTO XCENT+FLAG*X,YCENT+FLAG*Y
260 NEXT A
270 FOR A=30 TO 1 STEP -1
280 X=A*SC:Y=OPTS(A)*SC
290 DRAWTO XCENT+FLAG*X,YCENT-FLAG*Y
300 NEXT A
310 FLAG=-FLAG
320 IF FLAG<>1 THEN 210
330 FLAG1=-FLAG1
340 IF FLAG1<>1 THEN 180
350 NEXT R
360 RUN "D:WAVER.BAS"
440 NEXT E
```

WAVER.BAS

```
360 GRAPHICS 56
370 CLR5=2
380 FOR D=1 TO 16
385 SOUND 2,D*10,10,8
390 FOR E=1 TO 16
395 SOUND 0,E*30,8,8
400 SETCOLOR 2,E,D
410 SETCOLOR 1,D,E
420 FOR F=1 TO CLR5*9
430 NEXT F
440 NEXT E
450 NEXT D
470 GOTO 360
490 END
```

Notes: Written by ATARI's own Todd Frye, the program LISTed above uses chaining (at line 360) to call the second (WAVER) routine. The program draws sets of almost (but not quite!) concentric circles using high resolution graphics mode (Mode 8). When the Moire pattern of circles is completely drawn, the program chains to WAVER, which changes the colors and brilliances of the hues. We have taken the liberty of adding a very mechanical-sounding series of sound effects to Todd's original program. Hope you enjoy it!

Note again that chained programs are completely independent, and the first program is no longer in RAM when the second is LOADed. Line numbers need not be related in any way.

7.3 Hints and Ideas

Save RAM as needed by storing data in disk files and bringing it in to RAM only as needed! You can trade time for the equivalent of more memory if you let the disk drive be your data storage medium instead of writing your programs with (very) memory-expensive matrices and arrays! This partial program example using sequential access to find a particular record will xxxxx illustrate this point:

```
100 FOR X = 1 TO 100
110 INPUT (or GET) #1, Y
120 IF Y = 10344 THEN 1000
130 NEXT X
1000 PRINT "THE VALUE OF Y IS ";Y
```

See how we can use the disk drive to do a search for a particular record. The technique is very much like searching through a RAM array. Sure it's slower, folks, but what do you want for nothing? Egg in your beer?

7.4 We Saved the Best for Last!

Boy have we got a deal for you! How would you like not 1000, not 2000, not 4000, but 5384 extra bytes (mmmmm, good!) of RAM for your BASIC programs that don't need DOS or the menu to execute them? And its all absolutely free!

Try the following program like this. First "boot up" DOS in the normal way. Now type in the following program.

BASIC Routine To "Erase" DOS and save 5384 RAM Bytes

```
10 POKE 10,35
20 POKE 11,242
30 POKE 12,136
40 POKE 13,7
50 POKE 1804,48
60 POKE 1805,18
70 TRAP 90
80 X = USR(1928)
90 X = USR(40968)
100 PRINT FRE(0)        (line 100 for testing purposes only)
```

Notes: This little routine simply "wipes out" the portion of the DOS/FMS software (booted from diskette) that gives you the DOS Menu and related features. You can still RUN or LOAD disk files, access data files, etc. using BASIC statements in the normal way. However if you now type DOS, you will get the memo pad mode just as you would if you typed DOS with no disk drive connected!

Note: This procedure may not work with later versions of DOS, as the Disk Operating System is undergoing constant revision and updating to provide you with more and more advanced and convenience features! It has been tested on the operating system, FMS, and BASIC in use as of 10/30/79.

APPENDIX A : GLOSSARY WITH REFERENCES TO TEXT

Access: The method (or order) in which information is read from or written to diskette; see also Random Access and Sequential Access.

Address: A location in memory, usually specified by a two byte number in hexadecimal or decimal format.

Alphabetic: The capital letters A-Z. ~~(normally upper case).~~

Alphanumeric: The capital letters A-Z and the numbers 0 - 9, and/or combinations of letters and numbers. Specifically excludes graphics symbols, punctuation marks, and ~~the like.~~ other special characters

Argument: Computerese for the string or numeric quantity that is supplied to a function and then operated on to derive the value "returned" ~~by~~ with the function's subroutine.

Array: A one dimensional set of elements that can be referenced one at a time or as a complete list by using the array variable name and one subscript. Thus the array B, element number 10 would be referred to as B(10). Note that string arrays are not supported by BASIC, and that all arrays must be DIMensioned. A matrix is a 2 dimensional array. ~~See also Matrix.~~

ATASCII: The method of coding used to store text data. In ATASCII (which is a modified version of ASCII, the American Standard Code for Information Interchange), each character and graphics symbol, as well as most of the control keys has a number assigned to represent it. The number is a ~~two~~ one byte code (decimal 0 to ~~255~~ 255).

Backup DOS Disk: An exact copy of the original master DOS/FMS diskette. Refer to Chapter 2 for instructions on how to create backup copies of your master diskette. You should keep backups of your DOS/FMS master diskette

and of all important data diskettes.

Baud Rate: Signalling speed or speed of information interchange in bits per second.

Bit: Computerese for "binary digit". The smallest unit of memory that the computer uses, capable only of representing the values 0 or 1.

Boot: This is the initialization program that "sets up" the computer when it is powered up. At conclusion of the Boot (or after "booting up" as we like to say in Silicon Gulch), the computer is capable of loading and executing higher level (BASIC, in ATARI'S case) programs. The name comes from the fact that the booting process allows the computer to pull itself up "by its own bootstraps," ie programs that reference the disk are loaded from diskette!

Break: To interrupt execution of a program. ~~The statement in BASIC, "BREAK" or~~ Pressing the BREAK key causes a break in execution.

Buffer: An area in RAM used to hold data for further processing, input/output operations, or whatever. A temporary storage area for data.

Byte: The smallest addressable unit of memory. It usually consists of 8 consecutive bits, each of which may be either 0 or 1. Thus the various permutations and combinations of bit values allow the storage of 256 different values in one byte (one value at a time, of course!).

Close: To terminate access to a disk file. Before further access to the file, it must be opened again. See Open.

Data: Computerese for information stored or transferred within the computer or between the computer and its peripheral devices such as the disk drive(s).

Debug: To isolate and eliminate errors from a program. Also to swat flies.

Decimal: Capable of assuming any one of ten conditions, such as the digits from 0 to 9. Decimal (base 10) numbering is the usual system used by humans (since we have ten fingers). Decimal units are stored in binary coded decimal format in the computer. See Bit, Hexadecimal, and Octal.

Default: A condition or value that exists or is caused to exist by the computer when you don't tell it to do something else. For example, unless you tell it to use a certain graphics mode, your computer will default to GRAPHICS 0.

Delimiter: A character that marks the start or finish of a data item but is not a part of the data. The quotation marks (") are used by most BASIC systems for this purpose.

Destination: The device or address that receives data during an exchange of information (and especially an I/O exchange). For example, when we SAVE a program onto a diskette, the diskette is the destination, while the source is RAM.

Device:

Directory: A listing of the files contained on a diskette.

Drive Specification or Drivespec: Part of the filespec that tells the computer which disk drive to access. If this is omitted, and if more than one drive is connected, the computer will generally search through all diskettes, starting with drive #1, until the desired file is found.

Drive Number: An integer (whole number) from to that specifies the drive to be used. Drive is the one closest to the computer electrically, and must contain the master diskette with the DOS/FMS.

End of File: A marker that tells the computer that the end of a certain file on disk has been reached. Beyond the marker, the file may contain "garbage".

Entry Point: The address where execution of a machine-language program or routine is to begin. Also called the transfer address.

File: An organized (we hope!) collection of data related in some way. A file is the largest grouping of information that can be addressed with a single command or statement. For example, a BASIC program is stored on diskette as a particular file, and may be addressed by the statements SAVE or LOAD (among others).

Filename: The alphanumeric characters used to identify a file. A total of 8 numbers and/or letters may be used, the first of which MUST be a letter.

Filename Extender: From 0 to 3 additional characters used following a period (required if the extender is used) after the filename. For example, in the filename PHONLIST.BAS , the letters "BAS" comprise the extender.

Filespec: A sequence of characters which specifies a particular disk file under DOS/FMS. See chapter , and the BASIC Reference Manual, Chapter 5.

Format: To organize a new or magnetically (bulk) erased diskette into tracks and sectors. When formatted, each diskette contains 40 circular tracks, with 18 sectors per track. Each sector can store up to 128 bytes of data.

Garbage: A random collection of data bits which are meaningless to the computer. The famous "GIGO" Law of computer operation states simply, "Garbage in, Garbage out" which is a nice way of saying that computers don't make mistakes. Humans make mistakes. If you, the programmer, feed it garbage, it will most certainly reciprocate!

Hexadecimal or Hex: Capable of existing or containing up to 16 different conditions. Also refers to the hexadecimal (base 16) numbering system, which uses these characters as digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Remember that computers use the binary system (0 and 1 only). ~~"Coding"~~ Numbers in base 16 or Hex ~~is~~ are easily translatable to binary, and this provides a compromise that the computer can use, and which is not totally beyond the capabilities of human programmers (at least the ones here at ATARI seem to manage, although a couple of them are spending a lot of company time trying to grow six more fingers...) The engineer's approach is quite different: He wants to amputate 4 toes!

Input: To transfer data from outside the computer (say, from a diskette file) into RAM. Output is the opposite, and the two words are often used together to describe data transfer operations: Input/Output or just "I/O". Note the reference point is always the computer. Output always means away from the computer, while Input means into the computer.

Kilobyte or K: 1024 bytes of memory. Thus a 16K RAM capability actually gives us 16*1024 or 16,384 bytes. Again, this is computerese, for kilo is actually a prefix that means 1000 times.

~~Library Commands:~~

~~Logical Record: e~~

Machine Language: The instruction set for the particular microprocessor chip used, in ATARI's case the 6502. All higher-level languages such as BASIC must be translated ("interpreted") into machine language in order to allow computer execution.

Null String: A string which has a length of zero. For example, A$="" stores the null string (nothing between the quotation mark delimiters) in A$.

Object Code: Machine language derived from "source code", typically from assembly language.

Octal: Capable of being in or containing one of 8 states. The octal numbering system uses the digits 0 through 7. Address and byte values are ~~frequently~~ sometimes given in octal form. The octal numbering system is in common use on the tropical Octal Islands, where the natives have only four fingers on each hand.

OPEN: To prepare a file for access by specifying whether it will be accessed randomly or sequentially, whether an input or output operation will be conducted, and specifying the filespec and disk number. See BASIC Reference Manual, Chapter 5.

Parameter: Optional information supplied with an I/O statement that specifies how the statement is to operate.

~~Password: e~~

Sector: The smallest block of data that can be written to a disk file or read from one. Sectors can store up to 128 bytes. Also called a "physical record".

Resident System Program: That part of DOS/FMS that remains in RAM after "booting up". This is the operating system that calls in other DOS/FMS programs as required.

Sequential Access: Reading from or writing to a diskette starting at the beginning of the file, and proceeding in order toward the end. See also Random Access.

String: A sequence of characters that must be looked at character for character to find meaning. In other words, the string does not in and of itself comprise a value. For example, the number "1135" represents a specific quantity, (1000+100+30+5). But the same group of four digits, stored AS A STRING represents no value whatever, and cannot be used to compute mathematical quantities using the arithmetical operators. See the BASIC Reference Manual, Chapter 7.

Track: A circle on a diskette used for magnetic storage of data. Each track has 18 sectors, each with 128 byte storage capability. There are a total of 40 tracks on each diskette.

Utility: A diskette-stored special purpose file that consists of a program or routine. ~~The DOS utilities provided on the master diskette are:~~

Appendix C

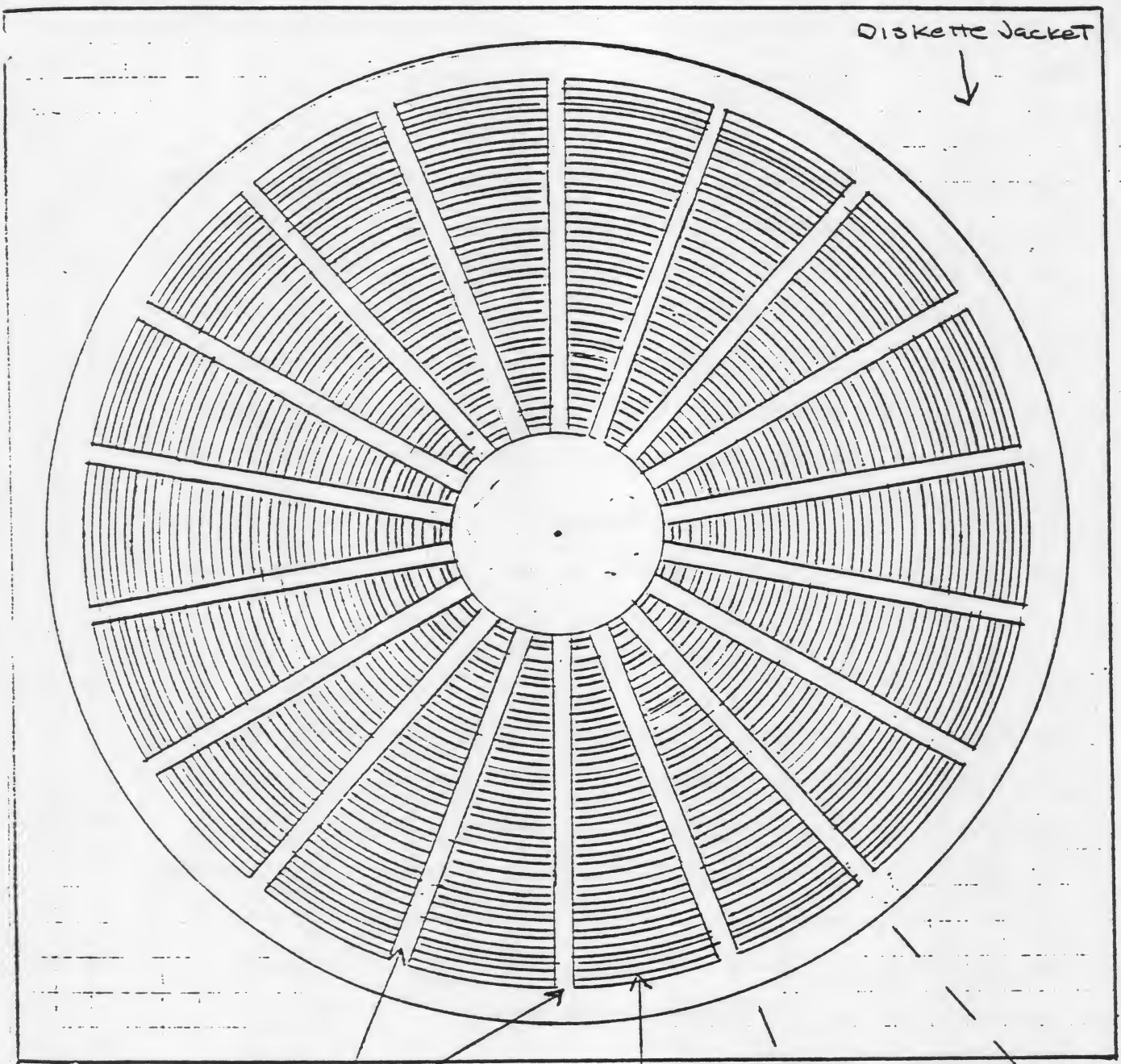
"PHYSICAL" FORMAT OF ATARI DISKETTES

The drawing in this Appendix shows the various sectors, tracks, and other physical format information. When a new Diskette is purchased, it is a totally blank disk of mylar material covered with a similar oxide material found on magnetic tape. The system of sectors, tracks, and breaks between them is magnetically produced when you use the DOS FORMAT DISK command. The WRITE NEW DOS command then stores the DOS software on the new disk within the physical structure produced during the formatting step.

A few specs and statistics: Each formatted ATARI diskette has 40 tracks. Each track is divided into 18 sectors, and each sector is capable of storing up to 128 bytes of data. Each individual byte on a disk is directly accessible using NOTE and POINT in BASIC.

40 tracks times 18 sectors per track times 128 bytes per sector yields a total storage capability of 92160 bytes. Of this, DOS itself consumes 7836 bytes, leaving approx. 83000 (or 83K) bytes for your storage use. As you can see, diskette space will seldom constrain you: It is much more likely to be the amount of RAM installed in the system that imposes the severest limitations upon the programmer. Bear in mind that program chaining is an excellent solution to this problem! So is the memory conservation trick described in Chapter 7.

ATARI DISKETTE FORMAT + SPECS



ATARI DISKETTE FORMAT AND SPECIFICATIONS SUMMARY

1. Each diskette Has:
   A. 40 Tracks
   B. 18 Sectors/Track less 11* for a total of 18 x 40 = 720 less 11 or 709 Sectors per diskette.
   C. Each Sector can store 128 bytes less 3 for overhead or 125 Bytes/Sector
   D. This gives a total disk capacity of 709 x 125 = 88,625 Bytes.

2. If the Diskette Has DOS:
   A. The DOS file itself uses 8K Bytes.
   B. This leaves a total of 80,625 Bytes storage capacity (Drive 1 or systems with only one drive connected.)

3. Diskettes on non-DOS Drives:
   A. Diskettes used on drives 2, 3, and 4 do not need the DOS file, and these have a total storage capacity of 88,625 Bytes.

4. A Four Drive System Has:

Drive No.	Capacity (Bytes)
1	80,625 (88,625 without DOS)
2	88,625
3	88,625
4	88,625
System Total:	346,500 Bytes

* 1 Sector used for Boot
8 Sectors used for Directory
1 Sector used for Voulme Table Content (Sector Usage Map)
1 Sector 720 not addressable and unused.

11 Sectors used by FMS (total of above)

HELPFUL HINT

Since the entire DOS, FMS, and all other programs carried on the Master Diskette under the file name DOS.SYS become RAM resident on booting up, once the system has been booted, the diskette that carries DOS may be removed and a formatted diskette without DOS may be substituted! This means that unlike most other microcomputer systems, your ATARI 800 personal computer system can get the full capacity per diskette possible, even with only one disk drive connected. This amounts to an extra 8K of storage space per diskette.

AND ANOTHER ONE...

Only operating one drive? Or two? Want to save a few bytes of RAM? Here's how: Since the DOS must be capable of controlling up to 4 drives, there are routines included in the DOS on your Master Diskette that you won't need if you're only operating one or two disk drives! By POKEing these routines out of the DOS while it is stored in RAM, rewriting the DOS onto your diskette, and then rebotting, you can save up to 397 bytes of RAM that would otherwise be used for the routines that control the drives you aren't using!

Caution! Use common sense any time you are POKEing! Check each POKE carefully before you hit RETURN. Remember that operating systems DO change, and it is possible that you may have one that already incorporates this routine or others so that the memory location to be POKEd has changed. So try this one out on a blank diskette to make sure it works perfectly on your particular system!

Here's the RAM saving technique, step by step:

- First "boot up" using your master diskette. Now remove the diskette and insert a totally blank one.
- Execute DOS item B (RUN CARTRIDGE) to get into BASIC, then POKE memory location 1802 with the value shown in the table below. (Use Direct Mode).
- Now do a PRINT PEEK(1802) and check that the value returned is the same number that you POKEd into that location. If it isn't, repeat the POKE step and check again.
- Go to DOS and use Menu item I (FORMAT DISK) to put the formatting onto your blank diskette. Now use item H (WRITE NEW DOS) to place the DOS.SYS file (with the new value of location 1802) on your new diskette.
- With the DOS file now on the new diskette, power down the computer to "empty out" the RAM. Power up (boot up) again with the new diskette installed.
- Now go to BASIC again and do a PRINT PEEK(1802),FRE(0) in direct mode. Location 1802 should show the new number you POKEd into it, and the free RAM should reflect a significant memory saving.

HERE ARE THE REQUIRED POKES

NUMBER OF DRIVES	POKE THIS VALUE INTO 1802	PRINT RRE(0)	BYTES SAVED (OVER 4-DRIVE CONFIGURATION)
1	1	21006	397
2	3	20867	258
3	7	20739	130
4	15*	20609	0

* Normal DOS configuration as supplied by ATARI